

Computer-
wissen

KOMPLETTER LEHRGANG:

ZAUBERN MIT DEM ZX 81

PROGRAMMIEREN
IN MASCHINENSPRACHE

Franzis' Bücher bei Knaur

Knaur

Taschenbuchausgabe
Droemersche Verlagsanstalt Th. Knaur Nachf. München
Lizenzausgabe mit freundlicher Genehmigung
des Franzis-Verlages, München

Umschlaggestaltung Adolf Bachmann
Umschlagillustration Ernst Jünger
Satz Appl, Wemding
Druck und Bindung Ebner Ulm
Printed in Germany · 1 · 8 · 985
ISBN 3-426-03794-7

1. Auflage

Kompletter Lehrgang: Zaubern mit dem ZX 81

In Maschinensprache programmieren

Mit zahlreichen Grafiken und Schaubildern

Knaur®



ISBN 3-426-03794-7 880

Inhalt

Software und Hardware

Laufzeit-Analyse

Ein verblüffendes Ergebnis erbrachte die Laufzeit-Analyse an zwei verschiedenen ZX-81-Computern: Der eine Computer ist fast doppelt so schnell wie der andere!

Durch geschicktes Programmieren ist es auch dem Einsteiger möglich, Programme »schneller« zu machen. Eine Voraussetzung dafür ist die Kenntnis über die Ausführungszeit einzelner Funktionen. So dauert z. B. beim ZX 81 das Potenzieren etwa 50mal länger als das Multiplizieren. Mit dem nachfolgend beschriebenen kurzen Programm läßt sich neben der Laufzeit arithmetischer und trigonometrischer Funktionen auch die ganzer Programmteile (z. B. Unterprogramme oder Schleifen) ermitteln.

Systemvariable ersetzt Stoppuhr

Zentraler Teil des Programms ist Zeile 60 *(Bild 1)*. Hier wird der Variablen X das Ergebnis der zu untersuchenden Funktion bzw. Operation zugewiesen. Um bei der späteren Zeitmessung auch die Dauer der Zuweisung selbst zu berücksichtigen, steht in Zeile 60 zunächst LET X = A. Diese Anweisung wird in einer Schleife 100mal ausgeführt um Abweichungen auszugleichen, die bei einmaliger Ausführung auftreten können.
Zur Zeitmessung wird die Systemvariable FRAMES verwendet (Adressen 16 436 und 16 437). Sie zählt die Anzahl der ausgegebenen TV-Bilder (50 pro Sekunde), indem ihr Wert, ausgehend vom Startwert 65 536, alle 20 ms um 1 verringert wird. Zum Zurücksetzen dieses internen Zeitgebers dient Zeile 40. Nach dem Ausführen der Schleife wird mit Zeile 80 der Zählerstand der Systemvariablen abgefragt. Zeile 90 übernimmt das Umrechnen des Werts in Millisekunden und normiert ihn so, daß mit X = A (Zeile 60) exakt der Wert 0 geliefert wird.
Da der Normierungsfaktor von Rechner zu Rechner abweichen kann, ist es möglich, daß nach dem Starten des Programms ein von 0 ver-

schiedener Wert auftritt! Dann ist das fünffache dieses Wertes vom Normierungsfaktor 65 034 (Zeile 90) abzuziehen. Gibt ein ZX 81 also

```
  10 LET A=PI/10
  20 LET A$=STR$ A
  30 LET X=0
  40 PAUSE 2
  50 FOR I=1 TO 100
  60 LET X=A
  70 NEXT I
  80 LET Z=PEEK 16436+256*PEEK 1
6437
  90 PRINT .2*(65034-Z)
 100 STOP
```

① **Listing »Laufzeit-Analyse«:** Damit läßt sich die Ausführungszeit einzelner ZX-81-Befehle ermitteln

z. B. -1.2 aus, dann ist 65 034 durch 65 040 zu ersetzen:
$65034 - (-1.2 \times 5) = 65040$.
Damit die benötigten Variablen schon vor der Zeitmessung bekannt sind, werden sie in den Zeilen 10 bis 30 angelegt und initialisiert.
Um nun die Rechenzeit z. B. für eine Addition zu ermitteln, ersetzt man Zeile 60 durch 60 LET X = A + A und startet das Programm mit RUN. Das Ergebnis ist diejenige Zeit, welche das Statement LET X = A + A länger dauert als das Statement LET X = A. Die beiden Anweisungen unterscheiden sich aber nur durch die Addition und den zusätzlichen Zugriff auf die Variable A. Das Programm berechnet also im wesentlichen die Rechenzeit für eine Addition (die zusätzliche Zugriffszeit kann dabei vernachlässigt werden).
Größere Programme können in die Zeilen 51 bis 69 geschrieben oder als Unterprogramm aufgerufen werden. Dann muß man aber auch die Zeit für GOSUB und RETURN berücksichtigen, und das macht eine erneute Anpassung der Normierungskonstante in der angegebenen Weise notwendig.
Die Resultate verschiedener Berechnungen sind in der *Tabelle* zusammengefaßt, wobei die farbig abgesetzten Werte mit dem ZX 81 der FUNK-SCHAU-Redaktion ermittelt wurden. Die genaue Ursache der erheblichen Abweichungen – die sich schon durch einen anderen Normierungsfaktor angekündigt haben – ist unbekannt. Da sich die Werte jedoch auch nach dem Überprüfen mit einer Stoppuhr als korrekt erwiesen haben, ist der Schluß zulässig: Es gibt unterschiedlich schnelle ZX 81! Wer sich nicht auf die modellunabhängige Aussage verlassen möchte, daß z. B. eine Addition nur halb so lange dauert wie eine Multiplikation, sollte deshalb die Analyse zum Ermitteln absoluter Zahlenwerte auf seinem ZX 81 nachvollziehen.

Die Resultate variieren etwas nach oben oder unten, wenn man die Variable durch Konstanten ersetzt (z.B. X=3**3 anstelle von X=A**A). Dies ist bei anderen Rechnern i.a. nicht so und durchaus einen Vergleich wert. Generell gilt aber, daß langsame Funktionen wie EXP, LN oder TAN auch auf anderen Rechnern langsam sind. Das Programm kann nur im SLOW-Modus zur Zeitmessung verwendet werden. Im FAST-Modus wird nämlich die Systemvariable FRAMES nicht verändert (keine Bildausgabe).
Abschließend sei noch darauf hingewiesen, daß auch die Zugriffszeiten auf die Variablen variieren. Und zwar werden sie um so größer, je später die Variable im Programm vereinbart wird (das hängt mit der Speicherorganisation zusammen). Dies hat aber kaum Bedeutung bei kurzen Programmen.

Michael Redmann

Tabelle: Ausführungszeiten einzelner Operationen und Funktionen. Der ZX 81 der FUNKSCHAU (farbig unterlegte Zeiten) ist fast doppelt so flott, wie der des Autors

−A	3,2 ms	1,8 ms
A+A	9,0 ms	5,0 ms
A*A	17,4 ms	9,6 ms
A/A	22,6 ms	12,4 ms
A**A	831,8 ms	456,0 ms
SQR A	835,6 ms	458,0 ms
PI	4,8 ms	2,6 ms
SIN A	288,6 ms	158,2 ms
TAN A	604,6 ms	382,6 ms
ASN A	1284,4 ms	704,0 ms
ATN A	429,0 ms	235,2 ms
INT A	8,4 ms	4,6 ms
ABS A	3,6 ms	2,0 ms
RND	85,8 ms	47,0 ms
PEEK A	8,8 ms	5,2 ms
VAL A$	428,6 ms	235,0 ms
CODE A$	8,8 ms	4,8 ms

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

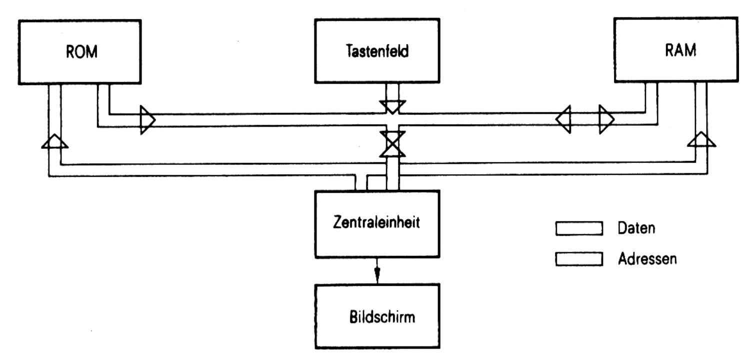
Teil 1: Daten lesen und verändern

Als Voraussetzung für das Programmieren in Maschinensprache bietet der ZX 81 zwei Basic-Befehle, die den Zugriff auf einzelne Bytes im Speicher erlauben.

Noch vor wenigen Jahren waren sogenannte Heimcomputer so beschaffen, daß man beim Programmieren nicht um Maschinensprache, also um die Befehlseingabe in hexadezimaler Form, herumgekommen ist. Moderne Heimcomputer verstehen hingegen Basic und können ohne jede Kenntnis von Maschinensprache gut programmiert werden. Aber die Maschinensprache lebt trotzdem weiter. Das zeigt sich bei einem Blick in Mikrocomputer-Zeitschriften oder beim Auflisten professionell geschriebener Programme und hat gute Gründe.
Als Einsteiger freilich steht man dem Maschinencode ratlos gegenüber, und die Bedeutung von Programmzeilen mit seltsamen REM-Kommentaren bleibt im Dunkeln. Wer sich aktiv mit seinem Heimcomputer beschäftigt und dessen Leistungsfähigkeit ausschöpfen möchte, wird aber gewiß nicht lange in der »Schreckstarre« verharren, sondern den Dingen auf den Grund gehen. Dabei will die Serie »Klartext für den ZX 81« Schrittmacherdienste leisten und leicht verständlich mit vielen praktischen Übungen das Programmieren in Maschinensprache verständlich machen. Da besonders Einsteiger angesprochen sind, wurde der ZX 81 als Übungsmodell ausgewählt.

Maschinenprogramme sorgen für Tempo

Maschinensprache ist eine elementare Sprache, auf die andere, höhere Programmiersprachen aufbauen. So wurde von den Entwicklern des ZX 81 erst einmal ein Maschinenprogramm geschrieben, das dem Computer die Programmiersprache Basic verständlich macht. Es wurde im ROM (Lesespeicher) des ZX 81 verankert und kann nicht verändert werden. Wenn Sie also ein Basic-Programm eingegeben haben und es durch RUN zum Laufen bringen, wird jedes Zeichen, jedes Schlüsselwort vom ROM übersetzt, d.h. interpretiert: Man nennt das Maschinenprogramm im ROM deshalb »Interpreter«.



① **Blockschaltung eines Computers:** Die Zentraleinheit kann mit 16-Bit-Adressen max. 65 535 8-Bit-Speicherzellen im RAM adressieren

Der Interpreter macht der Zentraleinheit im Computer verständlich, was die Basic-Programmschritte bezwecken sollen. Leider nimmt dieses andauernde Interpretieren viel Zeit in Anspruch, was sich besonders bei schnellen Spielen, bei bewegten Grafiken, beim Rechnen und überall dort, wo Zeit kostbar ist, störend bemerkbar macht. Es muß also gelingen, den Interpreter zu übergehen und gleich ein für den Computer unmittelbar verständliches Maschinenprogramm anstelle des Basic-Programms zu schreiben. Doch bevor wir damit beginnen können, ist etwas Theorie notwendig.

Die Hausnummern der RAM-Straße

Neben dem ROM hat der ZX 81 auch noch ein RAM (Arbeitsspeicher). Wie *Bild 1* zeigt, lassen sich hier im Gegensatz zum ROM die Daten auch verändern. Selbstgeschriebene Maschinenprogramme können also nur dort abgelegt werden. Doch wie sieht nun so ein Speicher aus?

Ein Speicher besteht aus vielen einzelnen Zellen, in die jeweils ein Bit paßt. Damit man sie gezielt ansprechen kann und nicht verwechselt, werden jeweils 8 Bit zu einer Gruppe zusammengefaßt (1 Byte) und mit einer Art Hausnummer versehen, nämlich einer Adresse. Ein gespeichertes Programm ist in solchen 8-Bit-Speicherzellen unterge-

bracht. Sehr wichtig ist, daß jedes Byte dann eine Dezimalzahl von 0 bis 255 darstellen kann ($255 = 2^8 - 1$; Eins wird abgezogen, weil die Zählung bereits mit Null beginnt).
Betrachten wir jetzt den Aufbau des Arbeitsspeichers (RAM) näher: Sein Bereich beginnt, wie aus *Bild* 2 ersichtlich, bei Adresse 16 384. Ohne Zusatzspeicher reicht er 1024 Byte (1 KByte) weit bis zur Adresse 17 407. Mit der 16-KByte-Erweiterung reicht der Arbeitsspeicher bis zur Adresse 32 767.
Jede Adresse wird nun beim ZX 81 (und bei vielen anderen Computern) durch zwei Bytes dargestellt, wobei die größte Dezimalzahl, die mit zwei Bytes erreicht werden kann, die Zahl 65 535 ist. Darauf kommt man, wenn man 256 quadriert und vom Ergebnis Eins abzieht (oder: $65\,535 = 2^{16} - 1$; der Exponent macht deutlich, daß der ZX 81 16 Adreßleitungen hat).
Doch zurück zum Arbeitsspeicher: Bei Adresse 16 384 beginnt der erste Teilbereich, der Bereich mit den Systemvariablen. Diese wollen wir vorerst beiseite lassen. Interessant wird es bei Adresse 16 509, denn dort beginnt der Bereich des Programmspeichers.

Ein Basic-Befehl deckt Speicherzellen auf

Die Sache mit der Adressierung von Speicherzellen wollen wir jetzt gleich in der Praxis erproben. Es ist daher Zeit geworden, den ZX 81 einzuschalten. Geben Sie danach ein:

```
10 REM  ABCDEF
```

Überlegen wir uns einmal, welche Adresse das »A« haben müßte. Wie gezeigt beginnt der Programmspeicher bei Adresse 16 509. Die ersten zwei Bytes werden für die Zeilennummer, die folgenden beiden Bytes für die Länge der Programmzeile beansprucht. Danach folgt das Byte für die REM-Anweisung. Erst dann, unter der Adresse 16 514, folgt der Code für das »A«. In *Bild 3* wird dieser Sachverhalt verdeutlicht.
Ob das »A« tatsächlich unter der angegebenen Adresse gespeichert ist, kann jeder selbst durch Anwenden des PEEK-Befehls nachprüfen. Durch PEEK n erfährt man nämlich vom ZX 81 den Wert des Bytes, welches unter der Adresse n steht. Geben Sie jetzt ein:

PRINT PEEK 16514

Der Computer muß jetzt den Code von »A«, nämlich 38 anzeigen (siehe Anhang A des ZX-81-Programmierhandbuchs von Sinclair). Selbstverständlich lassen sich auf diese Weise auch die übrigen Adressen abfragen.

Wie durch Geisterhand: die REM-Zeile wird verändert

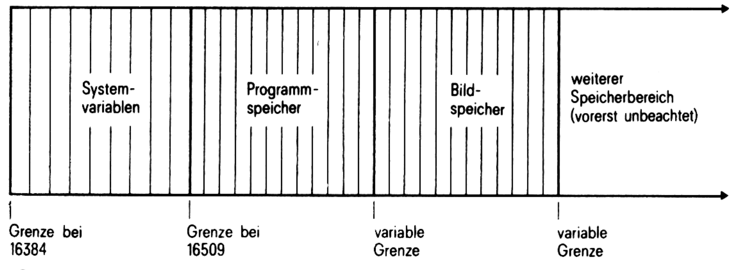
Es bringt einen natürlich nicht viel weiter, wenn man Inhalte von Adressen lediglich abfragen kann (in der Fachsprache heißt das »peeken«). Vielmehr wollen wir die Inhalte auch verändern. Dafür gibt es in Basic den POKE-Befehl. Durch POKE n,m kann man das Byte unter der Adresse n auf die Zahl m setzen.
Versuchen wir also, aus dem »A« in der REM-Zeile ein »G« zu machen. Die Adresse von »A« ist noch immer 16 514. Deshalb darf man getrost

POKE 16514, CODE »G«
oder
POKE 16514,44

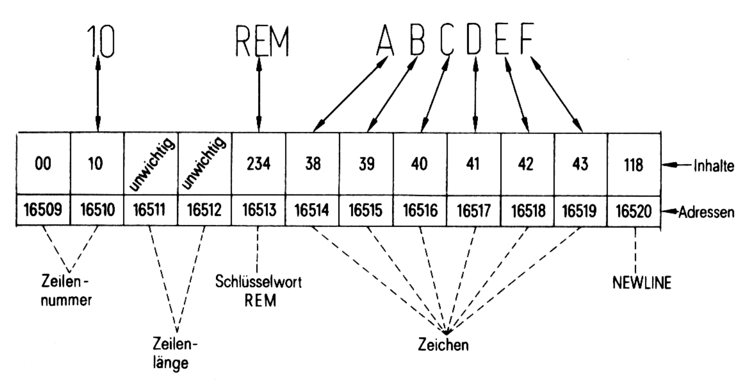
eingeben. Nach dem Auflisten wird die REM-Zeile die erwünschte Veränderung zeigen.
Versuchen Sie jetzt einmal selbständig, allein durch »poken« die REM-Zeile so zu ändern, daß auf dem Bildschirm

10 REM GHI PRINT Y

steht. Dabei ist zu beachten, daß PRINT im Bildspeicher Platz für mehrere Bytes beansprucht (fünf Buchstaben), der Befehl im Programmspeicher jedoch nur den Umfang von einem Byte hat. PRINT samt vor- und nachgestelltem Leerzeichen läßt sich deshalb mit dem Schlüsselwort-Code 245 in die REM-Zeile bringen.
Wer die Aufgabe gelöst hat, ist fürs erste fertig. Wer nicht, sollte alles nochmal durchackern.
Im nächsten Teil geht es mit Systemvariablen weiter. Zur Vorinformation ist dafür eine kleine »Hausaufgabe« ratsam: Blättern Sie Ihr Anleitungsbuch zum ZX 81 durch und lesen Sie das Kapitel mit den Systemvariablen und das mit dem hexadezimalen Zahlensystem. Außer-



② **RAM-Speicher des ZX 81:** Der Speicher ist in mehrere Sektoren eingeteilt



③ **Speicherbelegung:** So steht die Programmzeile »10 REM ABCDEF« im Programmspeicher des ZX 81

dem ist bald die Anschaffung eines Zusatzspeichers erforderlich, um die im Verlauf des Lehrgangs gegebenen Übungen nachvollziehen zu können: 4 KByte freier Speicherplatz sind dabei das Mindeste. Eine Bauanleitung für einen passenden Zusatzspeicher mit 6 KByte RAM (zusammen mit einem I/O-Port) finden Sie ab Seite 47.

Klaus Herklotz

Klartext für den ZX 81

Teil 2: Bedeutung der Systemvariablen

Nicht nur ein Programmierer kann Variablen Werte zuweisen – auch der Computer selbst kann das, wobei für ihn sogenannte Systemvariablen reserviert sind.

Im Arbeitsspeicher des ZX 81 befindet sich der Bildspeicherbereich direkt hinter dem Programmspeicherbereich. Wenn nun der Bereich des Programmspeichers durch Eingabe von Programmzeilen anwächst, dann verschiebt sich der Bildspeicherbereich gezwungenermaßen. Im Gegensatz zu der festen Adresse, die den Beginn des Programmspeichers markiert, ist die Grenze zwischen diesen beiden Speicherbereichen variabel.
Benutzer des Computers können jedoch den momentanen Stand der Grenze erfahren, weil zwei 8-Bit-Speicherzellen mit dieser Information belegt werden. Die beiden Speicherzellen befinden sich im Bereich der Systemvariablen und tragen den Namen D-FILE.

D-File beherbergt die Adresse des Bildspeicherbeginns

Die Systemvariable D-FILE belegt die Adressen 16396 und 16397 im Arbeitsspeicher. Der erste denkbare Schritt wäre, die Adressen zu lesen:

```
PRINT PEEK 16396, PEEK 16397
```

schreibt den Inhalt der beiden 8-Bit-Speicherzellen nebeneinander auf den Bildschirm. Wenn der Programmspeicher noch leer ist, sind das die Zahlen 125 und 64. Es stellt sich jetzt die Frage, was die beiden Zahlen bedeuten.
Beide Zahlen ergeben – richtig miteinander verknüpft – die Adresse, bei der der Bildspeicherbereich beginnt. Mikroprozessor-Architekten haben vereinbart, daß das hintere Byte das »meist signifikante« Byte

ist (Most Significant Byte, MSB). Man muß deshalb das Byte der höherwertigen Adresse (hier: 16397) mit 256 multiplizieren und zum Byte der niedrigeren Adresse (16396) addieren. Es ergibt sich $256 \times 64 + 125 = 16509$.
Zum Verständnis ein Beispiel: Jemand will seinem Gegenüber eine zweistellige Zahl mitteilen, kann aber immer nur eine Ziffer signalisieren. Beide haben deshalb vereinbart, daß der Empfänger der Zahlen immer zwei Ziffern hintereinander signalisiert bekommt, letztere mit 10 multipliziert und zur ersten addiert. Wird also die Ziffer 2 gefolgt von 6 signalisiert, so weiß der Empfänger, daß es sich um die Zahl 62 handelt.
Auf die gleiche Art geschieht dies beim ZX 81. Eine Adresse kann nicht in einer 8-Bit-Speicherzelle gespeichert werden, denn dazu ist sie mit ihren 16 Bit viel zu groß. Sie wird deshalb auf zwei 8-Bit-Speicherzellen aufgeteilt. Um die tatsächliche Adresse zu »konstruieren«, muß der Inhalt der hinteren Speicherzelle mit 256 multipliziert werden. Es ergibt sich deshalb aus den Zahlen 125 und 64 die Adresse 16509. Im Teil 1 war zu lesen, daß diese Adresse auch der unver-

Vorsicht beim Poken!

So reizvoll das Poken auch ist, so riskant ist es auch: Wird dem ZX 81 nämlich ein Byte an einer Speicherstelle aufgezwängt, deren ursprünglicher Inhalt für den internen Betriebsablauf des Computers maßgebend ist, dann protestiert der ZX 81, indem er verrückt spielt. Kuriose Bilder am Sichtschirm oder ein Löschen des Bildspeichers gepaart mit strikter Befehlsverweigerung sind das Ergebnis. Dem Computer schadet das nicht, aber seine Dienste können nur nach kurzfristigem Unterbrechen der Stromversorgung wieder in Anspruch genommen werden; dabei wird selbstverständlich auch das RAM gelöscht: Werden daher POKE-Befehle in ein längeres Basic-Programm eingebaut, dann sollte sicherheitshalber vor dem Starten mit RUN das Programm auf Magnetband gespeichert werden!

Geben Sie ein: POKE 16542,38
Nach dem zweiten NEW LINE zeigt der ZX 81 die kalte Schulter.

rückbare Anfang des Programmspeichers ist. Das ist jedoch nicht verwunderlich, denn der Programmspeicher ist noch leer; somit beginnt dort der Bildspeicher.
Die durch Abfragen der Systemvariablen D-FILE gewonnene Adresse besagt also, daß die nächste Adresse (hier: 16510) die Adresse der Bildspeicherzelle links oben in der Ecke des Bildschirms ist. Die Eingabe

POKE 16510, CODE »A«

läßt dort ein A auftauchen. Nun ist aber zu bedenken, daß der ZX 81 in der Grundversion mit 1 KByte Speicherumfang lediglich 1024 Speicherzellen zur Verfügung stellt. Der ZX 81 ist deshalb in bezug auf den Bildspeicher zur Sparsamkeit erzogen worden: Wenn der RAM-Speicherplatz unter $3\frac{1}{4}$ KByte liegt (siehe Sinclair-Handbuch Kapitel 27), besteht bei leerem Bildschirm der Bildspeicher nur aus den 33 NEW-LINE-Codes vom Zeilenende *(Bild 1)*.
Ohne Zusatzspeicher kann man zwar mit dem PRINT-Befehl arbeiten (der Bildspeicher wird dann um die erforderlichen Speicherbytes erweitert!), nichts bringt den Computer aber dazu, durch POKE 16510, CODE »A« Platz für das A zu schaffen. Das A wird zwar noch geschrieben, doch wird dabei einer der NEW-LINE-Codes gelöscht, was den ZX 81 unweigerlich funktionsunfähig macht. Mit Zusatzspeicher (Bauanleitung voraussichtlich im nächsten Heft) erscheint das A ohne Wenn und Aber am Bildschirm.
Wie Bild 1 zeigt, befinden sich dann in jeder Zeile des Bildschirms 33 Bildspeicherzellen. (Achtung: Zählung beginnt in Bild 1 bei 0.) Sie sind fortlaufend numeriert; letztes Zeichen in jeder Zeile ist NEW LINE. Das Fernsehbild nimmt also mit Zusatzspeicher bei 24 Zeilen $33 \times 24 = 792$ Byte Speicherplatz in Anspruch.
Um z. B. in der rechten unteren (auch durch PRINT AT noch erreichbaren) Ecke ein A zu schreiben, ist die Eingabe von

POKE 725 + PEEK 16396 + 256 * PEEK 16397,38

erforderlich ($725 = 33 \times 22 - 1$; A $\hat{=}$ Code 38). Damit wird klar, daß nicht nur mit PRINT AT Punkte des Bildschirms erreichbar sind, sondern auch durch ein direktes Belegen der Bildspeicherzellen. Dies ist später für die Maschinensprache von Bedeutung.

Aufspüren der aktuellen PRINT-Position

Die nächste wichtige Systemvariable heißt DF-CC. Sie gibt die Adresse der (letzten) PRINT-Position im Bildspeicher an und wird in den Adressen 16398 und 16399 abgelegt.
Mit dem folgenden Programm läßt sich feststellen, welche Adresse eine Bildspeicherzelle hat, von der man die Zeile und Spalte kennt:

```
10 PRINT AT 0,0:
20 LET A=PEEK 16398+256*PEEK 16399
30 PRINT AT 10,10; A
```

Mit Zeile 30 wird die Adresse (A) der Bildspeicherzelle für die PRINT-Position nullte Spalte und nullte Zeile ausgegeben; der Wert ist nicht 16510, da jetzt der Programmspeicher nicht mehr leer ist. Die Eingabe

```
10 PRINT AT 3,4;
```

verändert die PRINT-Position im Bildspeicher: Jetzt wird die Adresse der Zelle in der vierten Spalte der dritten Zeile ausgegeben. Durch

```
POKE A,38
```

wird an dieser Position ein A geschrieben.
Die praktische Bedeutung dieser Systemvariablen liegt aber nicht im Poken, sondern im Peeken: Es läßt sich nämlich der Inhalt sämtlicher Bildspeicherzellen ohne nennenswerten Aufwand lesen.
Das folgende Listing verdeutlicht das:

```
10 PRINT AT 3,4; »ABCDEF«
20 PRINT AT 3,4;
30 LET A=PEEK 16398+256*PEEK 16399
40 PRINT AT 20,0; PEEK A
```

Von dem Programm wird eine Buchstabenfolge ausgedruckt (Zeile 10). Danach wird die alte PRINT-Position wieder eingenommen (Zeile 20) und deren Adresse festgestellt (Zeile 30). Diese Adresse wird gelesen und der Inhalt – hier 38 ≙ A – ausgedruckt (Zeile 40). Um andere Speicherzellen zu lesen, muß z. B. nur die zweite Koordinate in Zeile 20 verändert werden.

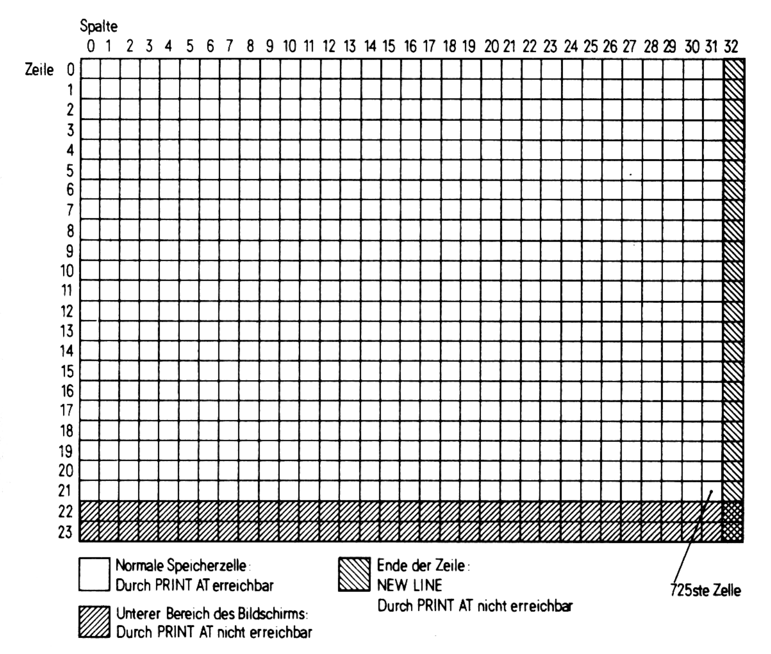
Der nutzbare Bildschirm wird größer

Die bisherigen Systemvariablen benötigen zum Unterbringen ihres Wertes zwei Adressen. Es gibt aber auch Systemvariablen, die sich mit einer Adresse begnügen: Sie werden deshalb Ein-Byte-Systemvariablen genannt.

Ein Beispiel dafür ist die Variable mit dem Namen DF-SZ. Sie hat die Adresse 16418 und ist ebenfalls für den Bildschirm zuständig. In DF-SZ ist die Anzahl an der Unterkante des Bildschirms normalerweise nicht verwendbarer Zeilen enthalten (belegbar sind nur 22 der 24 Zeilen).

Der Inhalt dieser Systemvariablen ist üblicherweise »2«, das heißt, es gibt zwei leere Zeilen am unteren Bildschirmrand. DF-SZ bietet jetzt die Möglichkeit, den nutzbaren Bildschirmbereich um diese zwei Zeilen zu vergrößern:

```
10 POKE 16418,0
20 PRINT AT 23,10; »ABCDEF«
```



① **Bildschirmorganisation:** Jede der 704 normalen PRINT-Positionen kann ein Zeichen aus dem Zeichenvorrat des ZX 81 aufnehmen

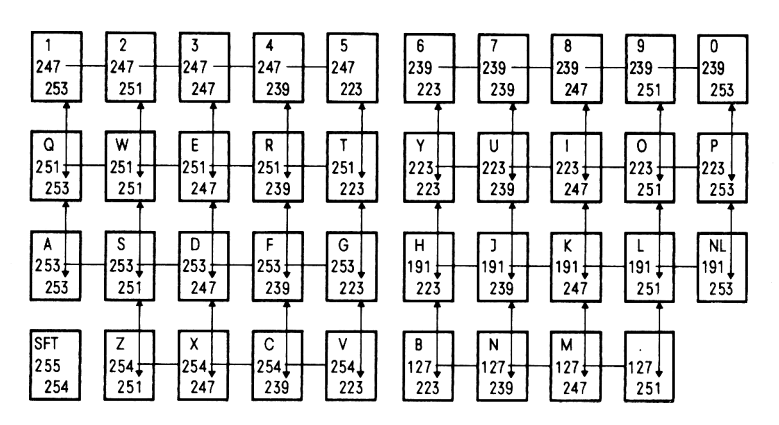
Auf die gleiche Weise läßt sich der nutzbare Bildschirmbereich verkleinern, was aber nicht sinnvoll ist.
Im Grunde ist diese Systemvariable unwichtig für spätere Maschinenprogramme. Sie soll nur stellvertretend für die Systemvariablen gezeigt werden, mit denen der Benutzer des Computers das System selbst verändern kann.

Welche Taste wurde zuletzt gedrückt?

Bis jetzt haben wir uns mit den Systemvariablen zur Datenausgabe beschäftigt. Es folgt nun eine zur Dateneingabe über das Tastenfeld: Ihr Name ist LAST-K. Sie ist in den Adressen 16421 und 16422 enthalten und zeigt an, welche Taste gerade gedrückt wird:

```
10 PRINT AT 0,0; PEEK 16421, PEEK 16422
20 GOTO 10
```

Wenn eine Taste gedrückt wird, ändern sich die beiden ausgegebenen Zahlen. In *Bild 2* sind die Werte tabellarisch aufgelistet.



② **Wertetabelle für LAST-K:** Abhängig davon, welche Taste *allein* gedrückt wird, ändern sich die Bytes unter den Adressen 16421 und 16422 (Systemvariable LAST-K) in der gezeigten Weise. Andere Werte ergeben sich, wenn zwei Tasten gleichzeitig gedrückt werden (z. B. SHIFT-Ebene)

Vor allem ist diese Systemvariable wichtig für spätere Maschinenprogramme, denn mit ihr läßt sich jede gedrückte Taste lokalisieren. Hier eine Anwendung:
Es soll eine Programmzeile entworfen werden, die, in ein Programm eingefügt, dieses solange unterbricht, bis die Taste P gedrückt wird. Im Basic liegt die Lösung auf der Hand:

```
10 IF INKEY$ <> »P« THEN GOTO 10
```

Wenn man die Systemvariable LAST-K einsetzt, sieht die Lösung so aus:

```
10 IF PEEK 16421<>223 OR PEEK 16422<>253 THEN GOTO 10
```

Wenn die Systemvariable LAST-K richtig eingesetzt wird, könnte auf die Funktion INKEY$ durchaus verzichtet werden!

Systemvariable zur Zeitmessung

Außer den bisher vorgestellten Systemvariablen gibt es noch gut 30 andere, die für Maschinensprache größtenteils unwichtig sind.
Eine noch interessante Systemvariable trägt den Namen FRAMES und ist in den Speicherzellen 16436 und 16437 zu finden. FRAMES kann für Zeitmessungen verwendet werden. Beim Einschalten des ZX 81 nehmen die Inhalte beider Zellen den Wert 255 an. Danach wird 50mal je Sekunde das weniger signifikante Byte (also das Byte aus 16436) um Eins verringert. Ist es schließlich bei Null angelangt, so wird es wieder zu 255 und gleichzeitig wird das mehr signifikante Byte um Eins verringert. Wenn beide Bytes Null geworden sind, nehmen Sie wieder den Wert 255 an.
Das Ganze läßt sich so demonstrieren:

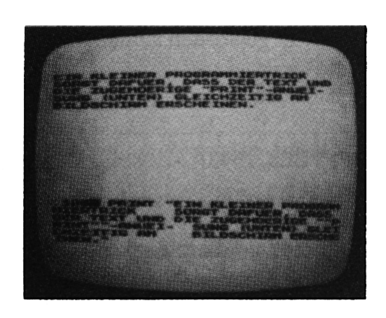
```
10 PRINT AT 0,0; PEEK 16436+256*PEEK 16437
20 GOTO 10
```

Zusammenfassend läßt sich über die Systemvariablen sagen, daß sie die Möglichkeit geben, etwas über den momentanen Zustand des Systems selbst zu erfahren, manche sogar die Möglichkeit geben, das System zu verändern. In Maschinenprogrammen helfen sie beim Schreiben am Bildschirm und bei der Dateneingabe. Klaus Herklotz

ZX-81-Softwaretip:

Textkorrektur – leicht gemacht

Wenn ein Programm den Computer veranlaßt, mit PRINT-Anweisungen längere Texte am Bildschirm auszugeben, dann ist das Ergebnis auf dem Bildschirm meist alles andere als schön: Trennungsstriche fehlen, die Worttrennung erfolgt an der falschen Stelle und alle Zeilen beginnen am linken Bildschirmrand, egal wie lang sie sind. Von einem gegliederten Textaufbau kann also keine Rede sein. Da der Zeilenfall bei der PRINT-Anweisung völlig anders ist als der am Bildschirm wiedergegebene, gelingt das richtige Einfügen von Trennungsstrichen und Leerzeichen oft erst nach mehreren Anläufen.
Schneller geht es, wenn die Programmzeile mit der PRINT-Anweisung und die ausgeführte PRINT-Anweisung *gleichzeitig* am Bildschirm erscheinen, da sich die Auswirkung einer Textänderung dann sofort am Zeilenfall nachprüfen läßt *(Bild)*.
Das ist kein Problem, wenn nach der PRINT-Anweisung in der nächsten Programmzeile ein STOP-Befehl vorübergehend eingefügt wird. Dann bewegt man im Programmlisting den Cursor nach oben bis zur Zeilennummer der PRINT-Anweisung. Gestartet mit RUN wird nun das Programm bis zum STOP-Befehl ausgeführt, der Text der PRINT-Anweisung auf den Bildschirm geschrieben und die Meldung 9/xxx ausgegeben. Jetzt ist nur noch EDIT einzugeben, um auch die Programmzeile mit der PRINT-Anweisung auf den Bildschirm zu holen. Nun kann nach Herzenslust korrigiert werden. -ll



Textkorrektur: Mit einem kleinen Trick läßt sich gleichzeitig zur Textanzeige die entsprechende PRINT-Anweisung auf den Bildschirm holen

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

Teil 3: »Hex« ist keine Hexerei

Keine Angst, hier geht es nicht zum x-ten Male um Sinn und Zweck des hexadezimalen Zahlensystems, sondern um den praktischen Umgang mit Hex-Codes.

Wer sich von der Pike aufwärts mit dem hexadezimalen Zahlensystem beschäftigen will, sei auf eines der vielen Mikrocomputer-Grundlagenbücher verwiesen. Wir hier setzen Grundkenntnisse voraus, zumal sogar das ZX-Handbuch ein eigenes Kapitel diesem Problem widmet. Der sichere Umgang mit dem hexadezimalen Zahlensystem ist und bleibt freilich beim Programmieren in Maschinensprache von allergrößter Wichtigkeit, so daß nachfolgend verschüttetes Wissen anhand praktischer Beispiele aufpoliert wird.
Als Einstieg sei kurz angemerkt, daß zur Darstellung eines Bytes im dezimalen Zahlensystem drei Stellen notwendig sind. Diese drei Stellen werden aber recht unbefriedigend genutzt: Es werden nur die Zahlen 0 bis 255 benötigt, obwohl 999 als größte dreistellige Zahl möglich wäre. Im hexadezimalen Zahlensystem braucht man zum Darstellen eines Bytes nur zwei Stellen, die jedoch restlos genutzt werden: FF ist damit die größte Zahl, die zur Darstellung eines Bytes notwendig ist.

Umwandeln per Programm

Weil die hexadezimalen Zahlen große Bedeutung haben, aber der Computer beim Poken nur Dezimalzahlen annimmt (bei der Adresse und beim Byte), stellt sich das Problem der Umwandlung von einem Zahlensystem ins andere. Bei Ein-Byte-Zahlen ist dies einfach. Zur Not reicht dazu der Anhang A des ZX-Handbuchs.
Bei größeren Zahlen beginnen aber die Schwierigkeiten: Wie lautet z.B. die Zahl 16514 in hexadezimaler Schreibweise? Das ist wirklich eine harte Nuß. Ein Umwandlungsprogramm *(Bild 1)* erleichtert die Arbeit ungemein. Es ist daher empfehlenswert, das Programm auf Kassette aufzunehmen.

Zum Umgang mit dem Programm: durch Drücken der NEW-LINE-Taste wird es abgebrochen. Außerdem nimmt der Rechner nur fünfstellige Dezimalzahlen und vierstellige Hexadezimalzahlen an: Kleinere Zahlen müssen durch Anfangsnullen auf die richtige Stellenzahl gebracht werden.
Für einen ersten Test nehmen wir die Zahl 16514: Wenn das Programm fehlerfrei eingegeben wurde, muß »4082H« ausgegeben werden. Gemeint ist also die Hexadezimalzahl 4082 (Hex-Zahlen werden mit H oder h gekennzeichnet, um Verwechslungen mit Dezimalzahlen zu vermeiden).
In Zeile 110 wird der eingegebene String I$ (String: Zeichenfolge) auf die Länge hin überprüft: Nur Strings der Länge 5 werden zugelassen und auf dem Bildschirm ausgegeben. In Zeile 120 wird dann ermittelt, ob die letzte Stelle des Strings ein H ist. Bei 16514 ist dies nicht der Fall: Weiter geht's deshalb ohne Sprung. Die Variable I erhält nun den Wert von I$.
Mit Zeile 140 werden die hinteren beiden Stellen (der zukünftigen Hexadezimalzahl) ermittelt und von Zeile 280 in eine Hexadezimalzahl umgewandelt. Dieser Wert wird unter I$ gespeichert und anschließend werden die vorderen beiden Stellen umgewandelt (Zeilen 170 und 180). In Zeile 190 wird der Ergebnisstring zusammengesetzt und mit Zeile 200 ausgegeben.
Als nächstes der umgekehrte Fall: 4082H wird in 16514 umgewandelt. In Zeile 120 wird diesmal festgestellt, daß eine Hexadezimalzahl eingegeben wurde: es erfolgt ein Sprung nach 300. Dort ermittelt der Rechner wieder die ersten beiden Stellen und wandelt sie in Zeile 480 in eine Dezimalzahl um. Diese Zahl wird in Zeile 320 unter der Variablen I gespeichert. Danach werden die hinteren beiden Stellen ermittelt und umgewandelt. In Zeile 350 ermittelt der Rechner schließlich die endgültige Zahl I.
Sehr wichtig, vor allem zum Verständnis des nächsten Programms, ist die Zeile 480: Dort wird ein String der Länge Zwei (mit einer Ein-Byte-Hexadezimalzahl) in eine Dezimalzahl umgewandelt. Gehen wir davon aus, daß es sich dabei um die Zahl FAh handelt.
Die Zahl FAh hat zwei Stellen, eine mehr signifikante (das F) und eine weniger signifikante (das A). Zuerst wird in der Zeile 480 der Code der mehr signifikanten Stelle ermittelt; er ist 43. Davon wird 28 abgezogen und man erhält die entsprechende Dezimalzahl für F, nämlich 15. Da es sich um die mehr signifikante Stelle handelt, muß man das Ergebnis mit 16 multiplizieren.

```
100 INPUT I$
105 IF I$="" THEN STOP
110 IF LEN I$<>5 THEN RUN
115 PRINT I$;" = ";
120 IF I$(5)="H" THEN GOTO 300
130 LET I=VAL I$
140 LET Z=I-256*INT (I/256)
150 GOSUB 280
160 LET I$=Z$
170 LET Z=INT (I/256)
180 GOSUB 280
190 LET I$=Z$+I$+"H"
200 PRINT I$
210 RUN
280 LET Z$=CHR$ (28+INT (Z/16))+CHR$ (Z-16*INT (Z/16)+28)
290 RETURN
300 LET Z$=I$(1 TO 2)
310 GOSUB 480
320 LET I=Z
330 LET Z$=I$(3 TO 4)
340 GOSUB 480
350 LET I=256*I+Z
360 PRINT I
370 RUN
480 LET Z=16*(CODE Z$(1)-28)+CODE Z$(2)-28
490 RETURN
```

① **Umwandlungsprogramm:** Damit werden Hexadezimal- bzw. Dezimalzahlen fürs jeweils andere Zahlensystem umgewandelt

Durch Umwandlung des A erhält man dessen dezimalen Wert 10, der zum vorhergehenden Ergebnis addiert werden muß. Das endgültige Ergebnis ist die Zahl 250.

Zurück zum Poken einer REM-Zeile

Die Methode eine REM-Zeile zu Poken (siehe Teil 1) hat einen entscheidenden Vorteil gegenüber der direkten Eingabe der REM-Zeile: Sogar die Fragezeichen-Symbole der nicht benutzten Codes können in der REM-Zeile erscheinen. Wie sonst könnte eine Speicherzelle den Inhalt 96 erhalten, wenn nicht durch Poken?
Das Poken soll jetzt vereinfacht werden. Ausgangspunkt ist die Zeile
10 REM 00000000
Diese Zeile soll zu
10 REM BDAFGHIK
verändert werden. Natürlich kann wieder jede Speicherzelle einzeln gepoked werden. Leichter ist es aber, eine Schleife zu programmieren und die Arbeit dem Computer zu überlassen.

Bild 2 zeigt das Programmlisting: In A$ sind die hexadezimalen Codes der Zeichen BDAFGHIK in der richtigen Reihenfolge enthalten. Danach wird die Anfangsadresse auf 16514 gelegt, das ist die Adresse des ersten Zeichens in der REM-Zeile. In der Schleife ab Zeile 40 werden die einzelnen Hexadezimal-Codes aus A$ umgewandelt und eingeschrieben.
Der hintere Teil der Zeile 50 hat große Ähnlichkeit mit der Zeile 480 des letzten Programms, er hat auch dieselbe Funktion. Soll eine andere Zeichenfolge in der REM-Zeile erscheinen, so müssen nur die Codes im String verändert werden.
Spätestens jetzt stellt sich die Frage, was die REM-Zeile mit dem Maschinencode, um den es letztendlich geht, zu tun hat.

Maschinenprogramme haben einen eigenen Startbefehl

Wie das Wort »Maschinencode« schon sagt, handelt es sich bei dieser Art des Programmierens um ein Programmieren mit Codes. Das heißt, daß jeder Befehl in Maschinensprache nicht durch ein Schlüsselwort (Keyword) wie in Basic, sondern durch eine Hexadezimalzahl von der Größe eines Bytes dargestellt wird.
In einer REM-Zeile sind Zeichen enthalten, deren Codes ebenfalls den Umfang eines Bytes haben. Es ist daher vorstellbar, ein ganzes Maschinenprogramm in eine REM-Zeile zu schreiben. Hier kann man schon den ersten Vergleich von Maschinensprache mit Basic anstellen: In Basic erfolgt die Eingabe eines Programmes durch Eintippen von Programmzeilen. In Maschinensprache dagegen wird ein Programm gerne durch Poken einer REM-Zeile eingegeben. Weiterhin wird ein Basic-Programm durch den Befehl RUN zum Laufen gebracht. Wie aber bringt man ein Maschinenprogramm (was das genau ist und wie es aussieht ist noch unbedeutend!) zum Laufen?
Aus früheren Überlegungen heraus steht fest, daß das erste Zeichen nach dem REM-Befehl in Adresse 16514 steht, wenn die REM-Zeile die erste Programmzeile ist.
Das Maschinenprogramm beginnt also bei Adresse 16514. Die Möglichkeit dort ein Maschinenprogramm aufzurufen bietet die USR-Funktion:

LET Q = USR 16514

```
10 REM 00000000
20 LET A$="2729262B2C2D2E30"
30 LET ADR=16514
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE ADR+INT (N/2),16*(CODE A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
```

② **Poken einer REM-Zeile:** Dieses Hilfsprogramm erspart das mühsame Poken einzelner Adressen

```
10 REM 0000000000000000000000000000000000000000000000000000
20 LET A$=""
30 LET ADR=16514
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE ADR+INT (N/2),16*(CODE A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
70 STOP
80 SAVE "USR"
90 LIST
95 STOP
```

③ **Eingabe von Maschinenprogrammen:** Mit diesem Programm lassen sich beliebige Zeichen des ZX 81 in der REM-Zeile unterbringen. Nach dem Laden wird es automatisch aufgelistet

befiehlt dem Rechner, ein in Maschinensprache geschriebenes Programm ab Adresse 16514 auszuführen. Die Wirkung ist mit GOSUB 16514 in Basic zu vergleichen.

Jetzt fehlt nur noch das Maschinenprogramm selbst. In *Bild* 3 ist ein Basic-Programm aufgelistet, das die Eingabe von Maschinenprogrammen und deren Aufruf erleichtert. Es enthält bereits eine lange REM-Zeile für lange Maschinenprogramme und sollte deshalb durch GOTO 80 auf Kassette gespeichert werden.

Das erste Maschinenprogramm, das vorgestellt wird, soll nichts tun! Im Z-80-Maschinencode gibt es tatsächlich den Befehl »nichts tun« (englisch: *no op*eration). Er hat die Abkürzung »nop« und den Hexadezimal-Code 00. Im Programm (Bild 3) muß also A$ = "" zu A$ = "00" verändert werden. Durch RUN wird das »Programm« in die REM-Zeile übernommen und durch LET Q = USR 16514 zum Laufen gebracht.

Hier gibts gleich die erste Überraschung: Der Computer steigt aus! Warum das so ist und wie man es vermeiden kann, verrät der nächste Teil. Klaus Herklotz

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

Teil 4: Datentransport mit »load«

In Basic weist der Befehl LET einer Variablen einen Wert zu. In Maschinensprache übernimmt der Befehl load diese Aufgabe.

Die letzte Folge endete mit einem Mißerfolg: Schon das einfachste Maschinenprogramm brachte den ZX 81 außer Kontrolle. Dies soll eine Warnung sein. Es darf nie ein Maschinenprogramm ohne Rücksprung abgeschlossen werden.
Durch LET Q=USR 16514 wurde unser Maschinenprogramm bei Adresse 16514 aufgerufen. Der Computer verläßt damit den Basic-Modus und interpretiert jedes Zeichen ab dieser Adresse als Maschinensprachebefehl. Erfolgt keine Rückkehr in den Basic-Modus, so weiß der Computer nicht mehr weiter und steigt aus oder liefert falsche Ergebnisse!

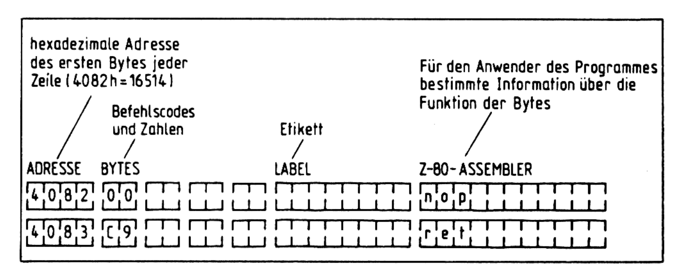
Zurück ins Basic

Im Z-80-Maschinencode gibt es für den Rücksprung ins Basic den Befehl »return«, was in Basic dem Rücksprung aus einem Unterprogramm gleichkommt. Das Befehlskürzel ist »ret« und der Hexadezimalcode C9.
Verwendet wird dieser Befehl vorerst als Abschluß eines Maschinenprogramms. Im Eingabeprogramm der letzten Folge muß A$="00" also zu A$="00C9" verändert werden. Durch RUN wird das Programm in die REM-Zeile gebracht und dann durch LET Q=USR 16514 zum Laufen gebracht. Diesmal bleibt der Computer bei der Stange und erledigt den Auftrag des »nichtstuns« ohne weiteres. Was geht hierbei im Computer vor?
Durch USR 16514 springt er in die Maschinenroutine bei Adresse 16514. Den Inhalt dieser 8-Bit-Speicherzelle faßt der Computer als Befehl auf: 00 heißt »no operation«. Das bedeutet, er kann sich der nächsten Speicherzelle zuwenden. Dort findet der Computer C9 und faßt es als return-Befehl auf: Rücksprung nach Basic.
Bei der Dokumentation längerer Maschinenprogramme wäre es sehr

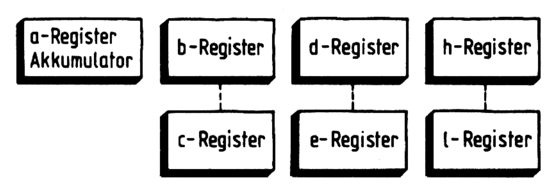
unübersichtlich, wenn einfach der String A$ = "...C9" mit hexadezimalen Bytes angegeben wird. Eine für den Anwender besser durchschaubare Form hat die allgemein übliche Notation von Maschinenprogrammen *(Bild 1):* Der hexadezimal angegebenen Adresse folgt ihr Inhalt (Byte). Der Übersicht halber können pro Zeile bis zu vier Bytes stehen. Warum das so ist wird später behandelt.
In der Spalte Z-80-Assembler steht schließlich die Übersetzung der Bytes in den sogenannten Assemblercode: nop heißt no operation und ret steht für return. Die Spalte Label ist noch unbedeutend.

Im Maschinencode kennt der ZX 81 Pseudo-Variablen

Im ZX-81-Basic kann der LET-Befehl jeder beliebigen Variablen einen Wert von −10E38 bis 10E38 zuweisen. Zum Beispiel weist LET B = 120 der Variablen B den Wert 120 zu.
In der Z-80-Maschinensprache stehen dem Benutzer sieben solcher Variablen zur Verfügung. Sie werden aber nicht Variable, sondern »Register« genannt *(Bild 2)*. Ein weiterer Unterschied zu Basic: Jedes



① **Listing zum Programm »Nichts tun«:** In dieser Form lassen sich Maschinenprogramme gut dokumentieren



② **Sieben Register des Z 80:** Im Maschinencode lassen sich den Registern Zahlen unmittelbar zuordnen. Das a-Register (Akkumulator) spielt bei den meisten Operationen eine maßgebende Rolle

ADRESSE	BYTES					LABEL	Z-80-ASSEMBLER
4082	06	00					ld b,00
4084	0E	01					ld c,01
4086	C9						ret

③ **Listing für das Laden des bc-Registerpaares:** Das Registerpaar wird mit der Zahl 0001 geladen

Register hat den Umfang eines Bytes, kann also nur Werte von 00 bis FFh aufweisen oder zugeordnet bekommen.
Ein Byte-Umfang ist zur Zahlenverarbeitung freilich etwas wenig. Es gibt daher die Möglichkeit, Register zu einem Registerpaar zusammenzufassen. Daraus entstehen dann das bc-Registerpaar, das de- und das hl-Registerpaar. Sie haben einen Umfang von zwei Bytes, können also Zahlen von 0000 bis FFFFh aufnehmen.
Auch der LET-Befehl hat in Maschinensprache einen anderen Namen; dort ist vom Datentransportbefehl load (laden) mit dem Kürzel »ld« die Rede. So ist z.B. »ld c, 01« gleichbedeutend mit LET C=1: Das c-Register erhält den Wert 01 zugewiesen. Für ld c, 01 gibt es aber keinen eigenen Code, wie z.B. C9 für ret (return). Es gibt nur den Code für ld c, N. Das große N steht stellvertretend für eine beliebige Zahl zwischen 00 und FFh. ld c, N hat den Code 0Eh. Die Zahl N selber steht in der nächsten 8-Bit-Speicherzelle. Um dem c-Register den Wert 01 zuzuweisen, muß also der String zu A$="...0E01..." erweitert werden.

Das bc-Registerpaar läßt sich auslesen

Maschinenprogramme in einer REM-Zeile werden z.B. durch LET Q=USR 16514, RAND USR 16514 oder PRINT USR 16514 aufgerufen. Als Verknüpfungsglied zwischen der Basic- und der Maschinenebene erhält dann der Ausdruck USR 16514 nach dem Rücksprung durch ret aus dem Maschinenprogramm, das bc-Registerpaar zugewiesen. Im Klartext heißt das: PRINT USR 16514 führt ein Maschinenprogramm ab Adresse 16514 durch und gibt danach den Inhalt des bc-Registerpaares am Bildschirm aus. Damit läßt sich der Inhalt dieses Registerpaares lesen.

Das erste richtige Maschinenprogramm

Jetzt wird ein Maschinenprogramm entworfen, daß das bc-Registerpaar mit 0001 lädt. Dazu sind zwei Schritte notwendig: Als erstes ist das b-Register mit 00 zu laden und danach das c-Register mit 01.
Für »ld c, N« ist der Code 0Eh. Für »ld b, N« ist er 06h. Eine Zusammenstellung der Maschinenbefehle enthält der Anhang A (Zeichenvorrat) des Sinclair-ZX-Handbuchs.
Das vollständige Listing des Maschinenprogramms zum Lösen der Aufgabe zeigt *Bild* 3. Demnach muß der String des Eingabeprogrammes zu A$ = "06000E01C9" verändert werden. Nach dem »Einpoken« durch RUN und dem Aufruf durch PRINT USR 16514 wird der Wert des bc-Registerpaares, nämlich »1« ausgegeben. Verfolgen wir wieder den Programmablauf aus der Sicht des Mikroprozessors:
PRINT USR 16514 schickt den Computer in den Maschinensprachemodus zur Adresse 16514. Ihr Inhalt ist der erste Befehl: 06 gebietet dem Computer, das b-Register mit der Zahl zu laden, die unter der nächsten Adresse zu finden ist. Das ist 00, womit die erste Operation bereits beendet ist. Der Prozessor faßt den Inhalt der nächsten Adresse wieder als Befehl auf: 0E veranlaßt ihn, das c-Register mit dem Inhalt der folgenden Adresse zu laden. Dort steht 01 und beendet den zweiten Schritt. Den Rücksprung nach Basic und den Ausdruck des bc-Registerpaares gebietet letztendlich C9 sowie der Aufruf des Maschinenprogramms durch PRINT USR.
Was aber wird ausgedruckt, wenn das c-Register mit 00 und das b-Register mit 01 geladen wird? Die Antwort liefert der Rechner. Vor Eingabe eines neuen Codes in Zeile 20 sollte die REM-Zeile sicherheitshalber wieder mit Nullen gefüllt werden, damit nicht Reste eines alten (längeren) Programms erhalten bleiben.

ld a, N	3E N
ld b, N	06 N
ld c, N	0E N
ld d, N	16 N
ld e, N	1E N
ld h, N	26 N
ld l, N	2E N

④ **Ladebefehle des Z 80:** Jedes der sieben Register hat zum Laden einer zweistelligen Hexzahl (N) einen eigenen Ladebefehl

r =	a	b	c	d	e	h	l
ld a, r	7F	78	79	7A	7B	7C	7D
ld b, r	47	40	41	42	43	44	45
ld c, r	4F	48	49	4A	4B	4C	4D
ld d, r	57	50	51	52	53	54	55
ld e, r	5F	58	59	5A	5B	5C	5D
ld h, r	67	60	61	62	63	64	65
ld l, r	6F	68	69	6A	6B	6C	6D

⑤ **Kopierbefehle des Z 80:** Jedes Register kann den Inhalt jedes anderen Registers übernehmen. Z. B. kopiert der Befehl 78 das b- ins a-Register

Genau wie die Register b und c, können auch die anderen Register mit einer Zahl geladen werden. *Bild 4* zeigt die Befehle mit den Codes. Für jede dieser Operationen werden zwei Speicherzellen benötigt: Eine für den Befehl und eine für die Zahl. Es handelt sich daher um Zwei-Byte-Befehle.

Kopierbefehle decken die übrigen Register auf

Mit den bisherigen Mitteln (Rücksprung mit Ausdruck des bc-Registerpaares) ist nicht feststellbar, welche Zahlen in den Registern d, e, h und l sowie im Register a (Akkumulator) stehen. Daher kommen als nächstes Befehle an die Reihe, mit denen einzelne Register in andere geladen werden, das heißt kopiert werden.
Der Befehl ld c, h lädt das h-Register ins c-Register. Dabei geht der Inhalt des h-Registers nicht verloren: Nach der Befehlsausführung ist das Byte des h-Registers sowohl im c-Register als auch im h-Register gespeichert! Der Befehl ld c, h wäre also mit LET C = H in Basic zu vergleichen. Eine Zusammenstellung der Kopier-Befehle zeigt *Bild 5*.
Mit Hilfe dieser Befehle können wir jetzt auch feststellen, welchen Inhalt z. B. das d-Register hat. Es muß nur das d-Register ins c-Register kopiert werden (b-Register mit 00 laden), denn der Rücksprung nach Basic bringt bekanntlich den Inhalt des bc-Registerpaares an den Tag (ret nicht vergessen). Genaueres steht im nächsten Teil.

Klaus Herklotz

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

Teil 5: Ladekommandos für Register

Weiter geht's mit dem Laden von Registern. Die CPU befolgt hierbei jede Menge von Ladekommandos.

Wie man einzelne Register mit Zahlen lädt, ist sicher noch vom vorherigen Teil bekannt. Es soll nun das hl-Registerpaar mit der Hexadezimalzahl 400E geladen werden. Ob diese Operation gelungen ist, läßt sich nicht so einfach nachprüfen. Dazu muß erst das hl-Registerpaar ins bc-Registerpaar kopiert werden und ein Rücksprung zum Basic erfolgen; schließlich wird immer nur der Inhalt des bc-Registerpaars dem Basic gemeldet!
Es empfiehlt sich, das Programm *(Bild 1)* über das in Teil 3 beschriebene Eingabeprogramm einzugeben und es durch PRINT USR 16514 abzurufen. Wenn alles glatt gegangen ist, wird 16398 (400Eh) ausgegeben.
Bis jetzt ist es uns nur möglich, ein Registerpaar mit *zwei* Schritten zu laden, wozu insgesamt vier Bytes notwendig sind. Schuld daran haben die 8-Bit-Ladebefehle. Wie *Bild 2* zeigt, ermöglichen 16-Bit-Ladebefehle das Laden eines Registerpaares in einem Schritt mit nur drei Bytes: Die beiden Speicherzellen nach der Speicherzelle des Befehlscodes enthalten dann die zu ladende Zahl. Dabei wird das Byte direkt

ADRESSE	BYTES				Z-80-ASSEMBLER
4082	2E	0E			ld l,0E
4084	26	40			ld h,40
4086	4D				ld c,l
4087	44				ld b,h
4088	C9				ret

① **Laden des hl-Registers:** Ob 400Eh tatsächlich geladen wurde, zeigt sich nach dem Kopieren ins bc-Register

hinter dem Befehlscode ins niedrigerwertige Register geladen (0E ins c-Register!) und das letzte Byte ins höherwertige Register (40 ins b-Register!). Nachfolgend die 16-Bit-Ladebefehle auf einen Blick:

ld bc, NN 01 ld de, NN 11 ld hl, NN 21

Schreiben Sie zur Übung das Programm aus Bild 1 mit einem 16-Bit-Ladebefehl.

»Peeken« in Maschinensprache

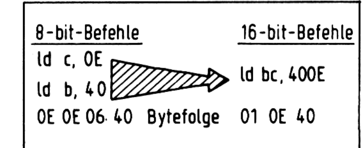
Jetzt beherrschen wir bereits die direkte Adressierung beim Ladevorgang, wobei Register direkt mit Zahlen oder dem Inhalt anderer Register geladen werden. Speziell für den Akkumulator gibt es jedoch weitere Ladebefehle:
Basic bietet die Möglichkeit, den Inhalt beliebiger Speicherzellen zu erfahren. Mittel dafür ist die PEEK-Funktion:

LET A = PEEK 16513

Diese Operation weist der Variablen A den Inhalt der Speicherzelle 16513 zu. In ähnlicher Weise funktioniert das auch in der Z-80-Maschinensprache. Das a-Register (Akkumulator) soll z. B. den Inhalt der Speicherzelle 16513 erhalten. In der Assemblerschreibweise sieht das so aus:

ld a, (4081h)

Die Klammer um die Zahl 4081h (16513) weist darauf hin, daß der *Inhalt* der Adresse 4081h gemeint ist. Genau wie bei den 16-Bit-Ladebefehlen sind auch hier drei Bytes notwendig: 3Ah als Kürzel von ld a, (NN) sowie die beiden Adreß-Bytes (wieder in vertauschter Reihenfolge). Das Maschinenlisting zu dieser Aufgabe ist *Bild* 3 zu entnehmen: Nach dem Rücksprung wird 234 ausgegeben. Und so ist es dazu gekommen: Nach dem Sprung in die Maschinenebene zur Adresse 4082h erhält der Computer die Anweisung, den Akkumulator mit dem Inhalt der Speicherzelle 4081h zu laden. In dieser Speicherzelle – das ist früheren Überlegungen zu entnehmen – befindet sich der Code des REM-Befehls, nämlich EAh (234). Danach wird das b-Register mit 00 und das c-Register mit dem Akkumulatorinhalt geladen, so daß vor dem Rücksprung die Zahl 00EAh (234) im bc-Registerpaar steht.



② **16-Bit-Ladebefehle:** Bei gleicher Wirkung wird gegenüber 8-Bit-Befehlen ein Byte eingespart

4082	3A	81	40		ld a, (4081)
4085	06	00			ld b, 00
4087	4F				ld c, a
4088	C9				ret

③ **Erweiterte Adressierung:** Jetzt wird der Akku mit dem Inhalt der Adresse 4081h geladen

Durch die Blume adressieren

Im wesentlichen gleicht die indirekte Adressierung der erweiterten Adressierung. Nur wird diesesmal die Adresse nicht direkt durch eine 16-Bit-Zahl dargestellt, sondern – wie der Name schon sagt – indirekt durch ein Registerpaar. So bewirkt z. B. der Befehl ld b, (hl), daß das b-Register mit dem Inhalt der Adresse geladen wird, die im hl-Registerpaar steht. Nachfolgend sind der Befehlsvorrat zur indirekten Adressierung sowie die dazugehörigen Codes aufgelistet:

ld a,	(NN)	3A	ld c,	(hl)	4E
ld a,	(bc)	0A	ld d,	(hl)	56
ld a,	(de)	1A	ld e,	(hl)	5E
ld a,	(hl)	7E	ld h,	(hl)	66
ld b,	(hl)	46	ld l,	(hl)	6E

Dem »Spielball« auf die Sprünge helfen

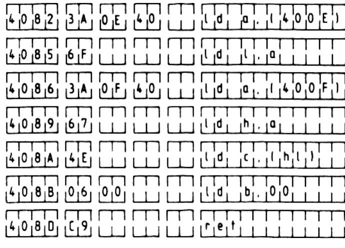
Mit den bisher gewonnenen Kenntnissen in Maschinensprache ist es möglich, einfache Probleme zu bewältigen: Nehmen wir an, es soll ein Programm entworfen werden, bei dem sich ein Spielball auf dem Bildschirm hin- und herbewegt. Wenn der Ball auf ein Hindernis trifft, soll er seine Richtung ändern. In Basic gibt es dann die Möglichkeit, die Koordinaten des Balles und die der Hindernisse zu vergleichen und bei Gleichheit eine Richtungsänderung einzuleiten. Dies nimmt jedoch, sobald die Zahl der Hindernisse größer wird, enorm viel Zeit und Speicherplatz in Anspruch. Einfacher ist es nachzuschauen, ob sich auf dem Feld, auf das der Ball als nächstes gelangt, ein Hindernis befindet. Wenn ja, dann soll der Ball die Richtung wechseln.
Zuerst die Lösung in Basic *(Bild 4):* Bis Zeile 270 werden die Variablen definiert und das »Spielfeld« gedruckt. Zeile 320 bringt die PRINT-Position auf das *voraussichtlich* nächste Feld. Dieses Feld wird in Zeile 340 in altbekannter Manier (siehe Teil 2) durch Abfragen einer Systemvariablen untersucht und dann eventuell die Bewegungsrichtung geändert. Der alte Spielball wird gelöscht (Zeile 390), der neue in Zeile 410 gedruckt und letztendlich die Koordination für den nächsten Durchgang festgelegt.
Damit sich das Lernen der Adressierungsarten auch gelohnt hat, versuchen wir jetzt, Zeile 340 durch eine Maschinenroutine zu ersetzen. Das Eingabeprogramm läßt sich jetzt natürlich nicht verwenden (hinzuladen), weil sonst das Basic-Programm gelöscht wird. *Bild 5* enthält

```
200 REM ANFANGSKOORDINATEN:
210 LET H=8
220 LET B=4
230 REM BEWEGUNGSRICHTUNG:
240 LET FH=1
250 LET FB=1
260 REM HINDERNISSE:
270 PRINT AT 2,4;"H";AT 7,1;"B";
    AT 17,13;"H";AT 12,16;"B"
300 REM ***HAUPTSCHLEIFE***
310 REM NAECHSTE PRINT-POSITION:
320 PRINT AT H+FH,B+FB;
330 REM FELDUNTERSUCHUNG:
340 LET Q=PEEK (PEEK 16398+
    256*PEEK 16399)
350 REM RICHTUNGSAENDERUNG:
360 IF Q=CODE "H" THEN LET FH=-FH
370 IF Q=CODE "B" THEN LET FB=-FB
380 REM DRUCKEN + LOESCHEN:
390 PRINT AT H,B;" "
410 PRINT AT H+FH,B+FB;"O"
420 REM KOORDINATNEN FIXIEREN:
430 LET H=H+FH
440 LET B=B+FB
450 GOTO 300
```

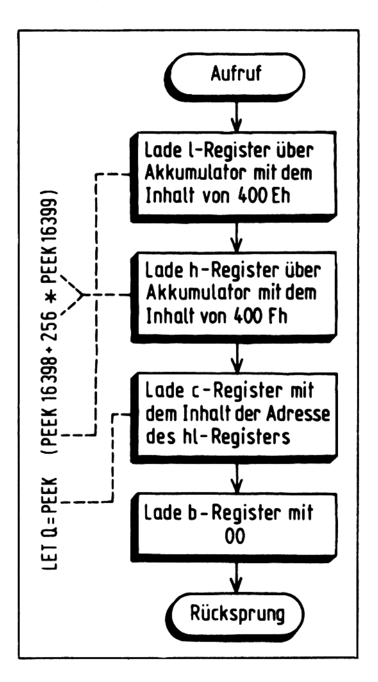
④ **Listing »Spielball«:** Ein Ball bewegt sich damit zwischen vier Pfosten noch recht gemächlich

```
10 REM 000000000000
20 LET A$="3A0E406F3A0F
   40674E0600C9"
30 LET ADR=16514
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE ADR+INT (N/2),16*(CODE
   A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
340 LET Q=USR 16514
```

⑤ **Maschinenroutine und Eingabeprogramm:** Der Ball wird schon deutlich schneller, wenn Zeile 340 des Basic-Programms unmittelbar auf Maschinenebene ausgeführt wird



⑥ **Maschinenlisting:** Mit zwölf Bytes wird die Wirkung der Basic-Programmzeile 340 auf Maschinenebene realisiert



deshalb außer dem Flußdiagramm auch noch ein komplettes Eingabeprogramm für den Maschinenteil. *Bild 6* zeigt das Maschinenlisting.

Ab Adresse 4082h wird der Akkumulator mit dem weniger signifikanten Byte der Systemvariablen DF-CC geladen. DF-CC ist in den Adressen 16398 (400Eh) und 16399 zu finden. Der Akkumulator wird dann ins l-Register kopiert.

Ab Adresse 4086h wird dann das mehr signifikante Byte von DF-CC ins h-Register gebracht. Auch hier dient der Akkumulator wieder als Zwischenspeicher. Nach diesen Operationen ist die PRINT-Position im hl-Registerpaar enthalten.

Unter Adresse 408A wird dem c-Register der Inhalt der Speicherzelle zugewiesen, deren Adresse im hl-Registerpaar steht: Das c-Register erhält somit den Code des Zeichens, das die PRINT-Position beschreibt. Zwecks Rückmeldung nach Basic erhält das b-Register noch den Wert 00, so daß im bc-Registerpaar der erwünschte Wert enthalten ist.

Klaus Herklotz

Klartext für den ZX 81

Teil 6: Auf Umwegen Adressieren

Neben direkter und indirekter Adressierung gibt es eine noch indirektere Adressierung. Zu ihrer Demonstration dient ein kleiner Abstecher in die Arithmetik.

Der Befehl zum indirekten Laden eines Registers läßt sich durch *ld r, (rp)* darstellen (siehe letzte Folge). Er legt fest, daß das Register r den Inhalt der 8-Bit-Speicherzelle rp erhält. Auch der umgekehrte Fall ist möglich: Eine Speicherzelle läßt sich mit einem Register-Inhalt laden. Der Befehl *ld (rp), r* ist dafür zuständig: Lade den Inhalt des Registers r in die Speicherzelle, die durch rp festgelegt wird. Der dazugehörige Befehlssatz mit den Codes ist nachfolgend aufgelistet.

ld (NN), a	32	ld (hl), c	71
ld (bc), a	02	ld (hl), d	72
ld (de), a	12	ld (hl), e	73
ld (hl), a	77	ld (hl), h	74
ld (hl), b	70	ld (hl), l	75

Wenn als Register r der Akkumulator verwendet wird, kann die Adresse der Speicherzelle auch direkt durch zwei Bytes angegeben werden. Es entsteht somit der Befehl *ld (NN), a*.
Als Übung versuchen wir, die Speicherzelle 16 513 (4081h) mit 245 (F5h) zu laden: Dazu wird einfach der Akkumulator mit F5h geladen und der Inhalt anschließend in die Speicherzelle 4081h gebracht *(Bild 1)*. Nach der Eingabe des Programmes und dem Aufruf durch LET Q = USR 16514 wird die REM-Zeile zur PRINT-Zeile. Eine Auflistung liefert den Beweis. Das ist eigentlich nicht überraschend, denn 245 ist der Code des PRINT-Befehls und 16513 die Adresse der Speicherzelle, in der der REM-Befehl stand.
Vergessen wir kurz einmal die Datentransportbefehle und fassen die Gruppe der arithmetischen Befehle ins Auge. Jetzt wird erstmal gerechnet.

Ein Ausflug in die Arithmetik

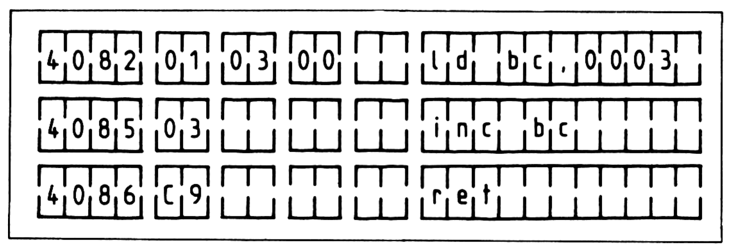
Der einfachste Arithmetik-Befehl erhöht den Inhalt eines Registers oder Registerpaares um Eins. Das Kürzel *inc* dieser Operation steht für increase (engl: erhöhen, vergrößern). Nehmen wir an, im Akkumulator steht die Zahl 03. Nach der Operation *inc a* weist der Akkumulator die Zahl 04 auf.
Ein weiterer Befehl vermindert den Inhalt eines Registers oder Registerpaares um Eins. Sein Kürzel *dec* steht für decrease (engl.: vermindern). Im Akkumulator soll wieder die Zahl 03 stehen. Nach *dec a* ist der Akkumulatorinhalt auf 02 reduziert worden.
Nachfolgend sind alle Inkrementier- und Dekrementier-Befehle aufgelistet. Es können sämtliche Register und Registerpaare, sowie der Inhalt einer im hl-Registerpaar stehenden Adresse erhöht bzw. vermindert werden.

inc a	3C	dec a	3D
inc b	04	dec b	05
inc c	0C	dec c	0D
inc d	14	dec d	15
inc e	1C	dec e	1D
inc h	24	dec h	25
inc l	2C	dec l	2D
inc (hl)	34	dec (hl)	35
inc bc	03	dec bc	0B
inc de	13	dec de	1B
inc hl	23	dec hl	2B

ADRESSE	BYTES				Z-80-ASSEMBLER
4082	3E	F5			ld a,F5
4084	32	81	40		ld (4081),a
4087	C9				ret

① **Speicherzellen belegen:** Die Speicherzelle 16513 erhält durch diese Maschinenroutine den Inhalt F5

Das Maschinenprogramm aus *Bild 2* zeigt den Befehl *inc bc* in der Praxis (Start mit PRINT USR ...). Dort wird zuerst das bc-Registerpaar mit 0003 geladen und dann der Inhalt um Eins erhöht. Nach dem Rücksprung wird somit 4 ausgegeben. Ersetzt man *inc bc* durch *dec bc,* so wird 2 ausgegeben.



② **Incrementierung:** Nach *inc bc* ist der Inhalt des Registerpaares bc nicht mehr 3, sondern 4

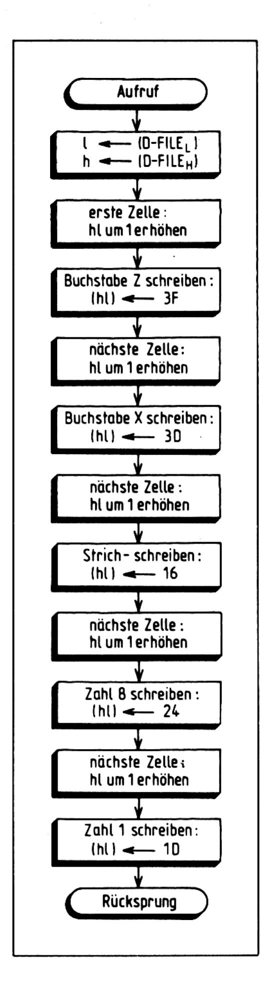
ZX 81 als Schnellschreiber

Viele ZX-81-Besitzer jammern über die Lahmheit ihres Rechners: »Man kann ja fast mitschreiben, wenn die Buchstaben nacheinander hingedruckt werden!« So schlimm ist es zwar auch wieder nicht, aber dennoch werden wir dem ZX auf die Sprünge helfen.

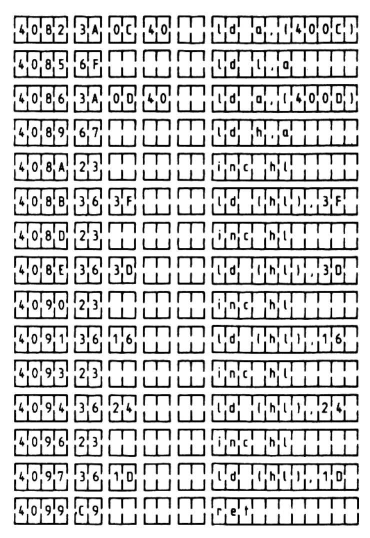
Zum Problem: Es soll ein kleines Maschinenprogramm geschrieben werden, das »ZX-81« auf den Bildschirm schreibt. Zuerst muß dazu die Adresse der linken, oberen Bildspeicherzelle ermittelt werden. Bei ihrer Ermittlung ist die Systemvariable D-FILE behilflich (vgl. Teil 2 in FS 11): Zuerst wird das höherwertige Byte von D-FILE ins h-Register und anschließend das niedrigerwertige Byte ins l-Register gebracht. Beide Male dient der Akkumulator als Zwischenspeicher.

Den weiteren Gang der Überlegung zeigt ein Flußdiagramm *(Bild 3):* Wenn nun der Inhalt des hl-Registerpaares erhöht wird, beinhaltet es die Adresse der ersten Bildspeicherzelle. Dort hinein laden wir den Code von »Z«, nämlich 3Fh, und der erste Buchstabe ist schon am Bildschirm geschrieben. Der Inhalt des hl-Registerpaares wird anschließend erneut erhöht und der Code des nächsten Buchstaben geladen. So geht es weiter, bis auch der letzte Buchstabe geschrieben ist.

Bild 4 zeigt das Maschinenlisting. Auf die gleiche Art und Weise läßt sich nun jeder beliebige Text mit ungemein hoher Geschwindigkeit



③ **Flußdiagramm:** Dieses Programm schreibt »ZX-81« am Bildschirm



4082	3A	0C	40		ld a,(400C)
4085	6F				ld l,a
4086	3A	0D	40		ld a,(400D)
4089	67				ld h,a
408A	23				inc hl
408B	36	3F			ld (hl),3F
408D	23				inc hl
408E	36	3D			ld (hl),3D
4090	23				inc hl
4091	36	16			ld (hl),16
4093	23				inc hl
4094	36	24			ld (hl),24
4096	23				inc hl
4097	36	1D			ld (hl),1D
4099	C9				ret

④ **Incrementieren nützlich angewendet:** Der Text »ZX-81« wird Schritt für Schritt in fünf Speicherzellen geladen

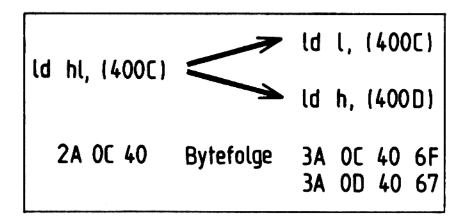
auf den Bildschirm bringen. Doch Vorsicht! Das 33ste Zeichen in jeder Zeile muß NEW LINE sein.

Bei der direkten Adressierung ist der Operand – gemeint ist damit das zu ladende Byte – direkt im Befehl enthalten. Die Z-80-Assemblerschreibweise lautet dann so: *ld r, r.*

Bei der indirekten Adressierung ist der Operand nicht bekannt, sondern nur seine Adresse, die der Inhalt eines Registerpaares ist. Daraus resultiert die Schreibweise *Id r, (rp):* Lade das Register r mit dem Inhalt der Adresse, die rp beschreibt.

Indirekter geht's nicht

Es gibt jetzt aber noch eine quasi »hyperindirekte« Adressierung: Man kennt dabei weder den Operanden noch seine Adresse, sondern nur die beiden aufeinanderfolgenden Adressen, in denen seine Adresse zu finden ist. Das klingt ziemlich kompliziert, aber genau den Fall hatten wir schon. Werfen wir noch einmal einen Blick auf Bild 4:
Von Adresse 4082h bis 4089h wird die Adresse der Bildspeichergrenze ermittelt. Die Adresse selbst ist noch unbekannt. Man kennt nur die beiden aufeinanderfolgenden Adressen, wo die beiden Adreß-Bytes zu finden sind. Um ein Registerpaar mit der tatsächlichen Adresse zu laden, sind dann vier Schritte mit insgesamt acht Bytes notwendig (zählen Sie selbst nach!). Ob dies wohl der Weisheit letzter Schluß ist? Sicher nicht, denn es geht viel kürzer!
Ausschlaggebend ist der Befehl *ld rp, (NN),* oder speziell für das Programm aus Bild 4: *ld hl, (400Ch).* Wie *Bild 5* zeigt, erfüllt der Befehl *ld hl, (400C)* eine Doppelfunktion. Einerseits wird das l-Register mit dem Inhalt der Speicherzelle 400Ch geladen. Auf der anderen Seite wird das h-Register mit dem Inhalt der nächsthöheren Speicherzelle, also mit dem von Adresse 400Dh, geladen.



⑤ **Eleganter Ladebefehl:** Mit nur drei Bytes wird der Inhalt von zwei Speicherzellen ins hl-Registerpaar geladen

ld bc, (NN)	ED 4B	ld (NN), bc	ED 43
ld de, (NN)	ED 5B	ld (NN), de	ED 53
ld hl, (NN)	2A	ld (NN), hl	22
$r_L \leftarrow (NN)$		$(NN) \leftarrow r_L$	
$r_H \leftarrow (NN+1)$		$(NN+1) \leftarrow r_H$	

⑥ **16-Bit-Ladebefehle zur indirekten Adressierung:** Wenn das hl-Registerpaar verwendet wird, sind nur drei Bytes notwendig

Allgemein heißt das: Der Inhalt der angegebenen Speicherzelle NN wird ins niederwertige Register geladen. Danach sucht sich der Prozessor selbständig die nächste Speicherzelle NN + 1 und lädt deren Inhalt ins höherwertige Register. *Bild 6* zeigt dazu den Befehlssatz.

Jetzt läßt sich das Programm aus Bild 4 verkürzen, indem die ersten acht Bytes durch 2A, 0C, 40 (ld hl, (400C)) ersetzt werden. Wie Bild 6 weiterhin entnommen werden kann, ist diese Form der indirekten Adressierung auch umkehrbar: *ld (NN), rp* lädt den Inhalt des niedrigerwertigen Registers (r_L) in die Speicherzelle NN, sucht sich die nächste Speicherzelle NN + 1 und lädt dort den Inhalt des höherwertigen Registers (r_H) hinein. Klaus Herklotz

Klartext für den ZX 81

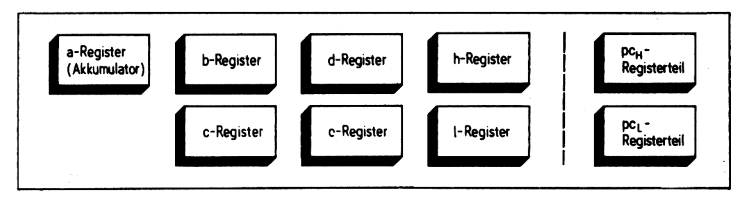
Teil 7: Jetzt wird im Kreis gesprungen

Die Datentransportbefehle mit load wurden im letzten Teil abgeschlossen. Ab jetzt geht es um Sprungbefehle und um alles, was dazu gehört.

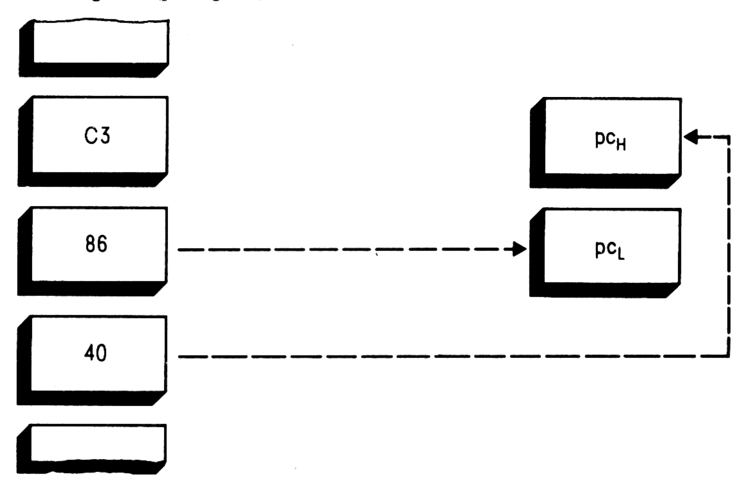
Wenn ein beliebiges Maschinenprogramm ausgeführt wird, dann werden die unter den einzelnen Adressen abgelegten Befehle nacheinander erledigt. Dies setzt voraus, daß der Prozessor immer weiß, welche Adresse als nächste an der Reihe ist.
Die Z-80-Entwickler haben deshalb der CPU einen Zähler spendiert, der stets die gerade aktuelle Adresse enthält und als Registerpaar realisiert ist. Die Rede ist vom Programmzähler oder pc-Register (program counter): Wann immer eine Adresse »abgearbeitet« worden ist, wird der Programmzähler inkrementiert (Wert um Eins erhöht). Wie *Bild 1* zeigt, erhöht sich damit unser Registervorrat auf neun Register.

Fast wie in Basic: der absolute Sprung

In Basic-Programmen folgt dem Schlüsselwort GOTO die Zeilennummer der Programmzeile, die als nächstes ausgeführt werden soll: Es erfolgt ein Sprung!
In Maschinensprache sind solche absoluten Sprünge ebenfalls programmierbar, nur läuft die Sache wieder unter einem anderen Namen ab: Hier kennt man den jump-Befehl mit dem Kürzel *jp*. Dieser Sprunganweisung folgt aber jetzt nicht die Programmzeile, sondern – wie in Maschinensprache üblich – die Adresse der Speicherzelle, zu der gesprungen wird. So führt der Computer z.B. durch *jp 4082* als nächstes die in der Speicherzelle mit der Adresse 4082h stehende Anweisung aus.
Der Operationscode von *jp NN* ist »C3.. ..«. Auch hier muß wieder das weniger bedeutende Adreß-Byte als erstes erscheinen. *Bild 2* verdeutlicht den Vorgang bei *jp NN:* Die beiden Adreß-Bytes werden in den Programmzähler geladen.



① **Registervorrat:** Auch der 16 Bit umfassende Programmzähler besteht aus zwei 8-Bit-Registern (pc-Register)



② **Absoluter Sprung:** Durch den jump-Befehl (Operationscode: C3) wird der Programmzähler pc mit der Zieladresse (hier: 4086h) geladen

Unser erstes Programm mit einem Sprungbefehl

Probieren wir den Sprungebefehl gleich an einem kurzen Maschinenprogramm aus. Recht viel Auswahl bleibt da kaum. Wir müssen uns, wie *Bild* 3 zeigt, vorerst damit begnügen, immer im Kreis zu springen.
Zuerst wird die Adresse der ersten Bildspeicherzelle ins hl-Registerpaar gebracht (4082h bis 4085h). In diese Zelle wird ein »X« geschrieben und gleich danach wieder gelöscht. Unter der Adresse 408A steht letztendlich der Operationscode des absoluten Sprunges gefolgt von der Zieladresse 4086h. Nach dem Aufruf dieses einfachen Programmes durch LET Q = USR 16514 flackert am linken oberen Bildschirm-

eck das X. Ein Nachteil läßt sich freilich nicht verschweigen: Nur durch Ziehen des Netzsteckers kann der Computer von der Flackerei erlöst werden!
Werfen wir noch kurz einen Blick zurück auf Bild 3. In der Z-80-Assemblerschreibweise ist die Zieladresse des Sprunges mit einem Label (Etikett) versehen. Dabei wird so vorgegangen, daß man für die Label-Spalte einfach einen passenden Namen erfindet (z.B. FLACK für flackern). Dann braucht auch der Befehlscode des Sprunges (bei uns in Adresse 408A) nicht von der Zieladresse selbst, sondern lediglich von ihrem Etikett gefolgt werden. Dies bringt schon fürs erste ein hohes Maß an Übersichtlichkeit.

Was heißt Assembler?

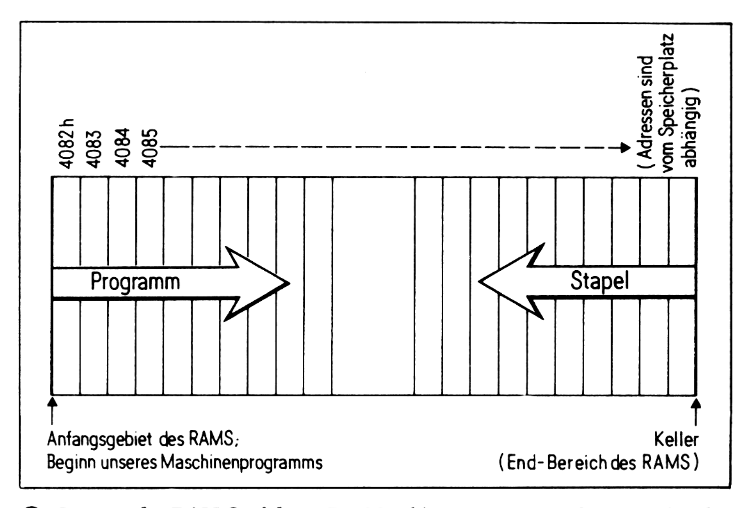
Die Hexcodes der Maschinenbefehle lassen sich alles andere als leicht merken, so daß das Schreiben oder Analysieren längere Maschinenprogramme sehr mühsam würde. Leichter geht das mit den symbolischen Abkürzungen (Mnemonics) der Maschinenbefehle (z.B. ret für Return). Ein mit symbolischen Abkürzungen geschriebenes Programm versteht der Prozessor freilich ebensowenig wie ein Basic-Programm, wenn ihm die Befehle nicht in Hexcodes übersetzt werden. In Basic hat diese Aufgabe z.B. ein Interpreter, wogegen Mnemo-Programme von einem *Assembler* übersetzt werden (der Assembler selbst ist meist ein Hexcode-Programm). Ein *Disassembler* übersetzt Hexcodes in die entsprechenden Mnemo-Abkürzungen (zur Programmanalyse). Programmieren in Assembler heißt, daß hier mit Mnemonics gearbeitet wird.

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4Ø82	2A	ØC	4Ø			ld hl,(4ØØC)
4Ø85	23					inc hl
4Ø86	36	3D			FLACK	ld (hl),3D
4Ø88	36	ØØ				ld (hl),ØØ
4Ø8A	C3	86	4Ø			jp FLACK

③ **Programm ohne Ende:** Durch eine mit jump gebildete Endlosschleife flackert ein Buchstabe im »Maschinensprachtempo«

Der »Stapel« als Datenspeicher

Jeder noch so billige Taschenrechner hat einen Zahlenwert-Speicher. Mit speziellen Tasten können Zahlen dort eingespeichert und zu einem späteren Zeitpunkt wieder abgerufen werden. Da unser Prozessor bislang nur sieben Register (ohne Programmzähler) zur Verfügung stellt, wäre ein solcher Speicher eine starke Sache.
Im Z 80 ist der Speicher als »Stapel« (engl.: Stack) gebaut. Wie *Bild 4* zeigt, befindet sich am einen Ende des RAM-Bereichs ein beliebiges Programm. Je länger das Programm ist, desto weiter reicht es in den RAM-Bereich hinein.
Fast ganz am anderen Ende des RAM-Bereichs befindet sich der »Keller« und zwar der Keller des Stapel-Speichers. Auf diesen Keller können wir nun Inhalte von Registerpaaren speichern. Dies geschieht wie bei einem Stapel Spielkarten: Die Karte, die als letzte auf den Stapel gelegt wurde, muß man später auch als erste wieder wegnehmen. Genaueres darüber im nächsten Teil. Klaus Herklotz



④ **Grenzen des RAM-Speichers:** Das Maschinenprogramm steht am Beginn des RAMs bei niedrigen Adressen. Der Stapel wächst dem Programm von der Obergrenze des RAMs entgegen. Für den Stapel ist die RAM-Obergrenze der Keller

ZX-81-Softwaretip:

Bremse für »Autostart«

Manche ZX-81-Programme im Handel haben die Eigenschaft, nach dem Laden von Kassette automatisch zu starten und sich dann nicht mehr unterbrechen zu lassen. Dies stört, wenn man so ein Programm näher untersuchen, kopieren oder abspeichern möchte. Das folgende Hilfsprogramm (Maschinencode) ermöglicht es, solche Programme so zu laden, daß sie nicht automatisch anlaufen:

```
1 REM 1234567890A
2 FOR A=16514 TO 16524
3 INPUT X
4 POKE A,X
5 NEXT A
```

Nach dem Start durch RUN sind die elf Zahlen 33, 102, 0, 229, 46, 8, 229, 55, 195, 67, 3 einzugeben und danach die folgenden Programmzeilen.

```
2 REM SPECIAL LOAD
3 FAST
4 RUN USR 16514
```

Jetzt kann Zeile 5 gelöscht und das Programm auf einer Kassette gespeichert werden. Nach dem Start des Programms durch RUN wird vom Maschinenprogramm die LOAD-Routine im ROM aufgerufen (Adresse: 835) und dadurch der Befehl LOAD"" ausgeführt. Nach erfolgreicher Beendigung des Ladevorgangs wird grundsätzlich mit Meldungscode 9 gestoppt. Hierfür verantwortlich sind spezielle Daten, die vor dem eigentlichen LOAD auf den Z-80-Stack gelegt worden sind.

Michael Schramm

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

Teil 8: Der Umgang mit dem »Stapel«

Daß der Stapel ein spezieller Bereich im RAM ist, wo man die Inhalte der Z-80-Register ablegen kann, wurde schon angesprochen. Jetzt kommt es darauf an, ihn zu nutzen.

In der Z-80-Maschinensprache werden Inhalte von Registerpaaren durch den push-Befehl gespeichert. So wirft z.B. *push bc* eine Kopie des bc-Registerpaares auf den Stapel, speichert also den Inhalt des bc-Registerpaars. Der Vorgang ist damit zu vergleichen, daß eine Karte auf einen Stoß mit Spielkarten gelegt wird.
Wie *Bild 1* zu entnehmen ist, geschieht das Einspeichern von Daten aus Registerpaaren in zwei Schritten. Zuerst wird das höherwertige Register und dann das niedrigerwertige Register auf den Stapel geworfen. Nun wissen wir also, wie solche Daten gespeichert werden. Es fehlt nur noch die Möglichkeit, die Daten wieder abzurufen.

So werden Daten vom Stapel »abgehoben«

Die Möglichkeit, dem Stapel Daten zu entnehmen, bietet der pop-Befehl. So holt sich z.B. *pop de* die obersten beiden Bytes vom Stapel und lädt sie ins de-Registerpaar. Wenn wir wieder an unser Kartenspiel denken, heißt das, daß wir die oberste Karte abheben.
Auch das Abrufen von Daten geschieht wieder in zwei Schritten *(Bild 2):* Zuerst wird das oberste Byte vom Stapel ins niedrigerwertige Register und dann das nächste Byte ins höherwertige Register geladen.

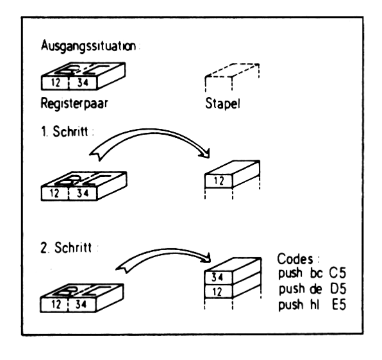
Der Stapel hilft uns beim Kopieren

Lösen wir jetzt ein altbekanntes Problem auf neue Weise: Es soll das hl-Registerpaar mit 0001 geladen und dann ins bc-Registerpaar kopiert werden. Der Ladevorgang des hl-Registerpaares soll dabei unverändert bleiben. Wie aus dem Maschinenlisting *(Bild 3)* ersichtlich, speichern wir dazu das hl-Registerpaar auf dem Stapel, Mit dem

nächsten Schritt wird dieses Datenpaar durch *pop bc* ins bc-Registerpaar geladen. Nach dem Aufruf des Maschinenprogrammes mit PRINT USR 16514 wird wie erwartet 1 ausgegeben.

Eine Gedächtnisstütze für den Prozessor

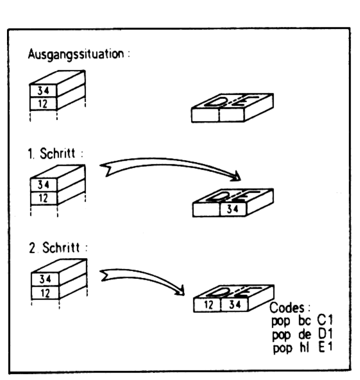
Genauso wie der Prozessor immer die gerade bearbeitete Adresse bei der Programmausführung kennen muß, so muß er auch wissen, bei welcher Adresse das obere Ende des Stapels liegt. Denn wie soll er sonst dort Daten aus Registerpaaren ablegen? Aus gutem Grund wurde deshalb ein »Stapelzeiger« installiert.
Der Stapelzeiger ist wieder als Registerpaar vorhanden, ähnlich dem Programmzähler. Gemeint ist der Stackpointer (Stapelzeiger) oder das sp-Register.
Immer wenn wir Daten auf dem Stapel ablegen (z.B. durch push), wird das sp-Register um 2 erniedrigt. Wem das nicht klar ist, der betrachte noch einmal Bild 4 aus Teil 7: Immer wenn Daten auf dem Stapel abgelegt werden, vergrößert das zwar den Stapel selbst, jedoch verringern sich die Werte der Adressen!
Wenn dem Stapel nun Daten entnommen werden (z.B. durch *pop*), dann wird das sp-Register um 2 erhöht. Beachtlich bei dem Ganzen ist, daß der Stapel selbst dabei nicht kleiner wird (Bild 2): Daten, die einmal eingespeichert wurden, werden nicht mehr gelöscht; es sei denn sie werden durch erneutes Einspeichern überschrieben. Diesem Umstand darf jedoch wenig Aufmerksamkeit beigemessen werden, weil er für den Programmieralltag nebensächlich ist.



① **Daten stapeln:** Die Ausgangssituation zeigt das bc-Registerpaar (Inhalt 1234) und das obere Ende des Stapels. Im ersten Schritt wird der Inhalt des höherwertigen Registers auf den Stapel gelegt und im zweiten Schritt der Inhalt des niederwertigen Registers

Der Stapel hilft bei Unterprogrammen

Wenn ein und dieselben Programmteile in einem Hauptprogramm mehrmals ausgeführt werden müssen, so empfiehlt es sich, Unterprogramme zu verwenden. In Basic werden Unterprogramme durch den GOSUB-Befehl mit nachfolgender Zeilennummer aufgerufen. Am Ende des Unterprogrammes steht dann der RETURN-Befehl, der einen Rücksprung ins Hauptprogramm zur Folge hat. In der Z-80-Maschinensprache kann die gleiche Wirkung mit den Befehlen *call NN* und *ret* (für return) erreicht werden.
Die Eingabe *call NN* bewirkt dasselbe wie *jp NN*, außer daß sich der Computer den Programmzählerstand vor dem Absprung (Rücksprungadresse) merkt. Die Rücksprungadresse wird dabei einfach oben auf den Stapel gelegt *(Bild 4)*.



② **Daten vom Stapel abrufen:** Die Ausgangssituation zeigt gestapelte Daten und das de-Register mit beliebigem Inhalt. Der erste Schritt lädt das zuoberst gestapelte Byte ins niederwertige Register, dann wird das zweite Byte ins höherwertige Register geladen

```
call NN  ──────► push pc
         ──────► jp NN

ret      ──────► pop pc
```

③ **Kopieren eines Registerpaares:** Zum Lösen des alten Problems führt der Weg diesmal über den Stapel

ADRESSE	BYTES				Z-80-ASSEMBLER
4082	21	01	00		ld hl, 0001
4085	E5				push hl
4086	C1				pop bc
4087	C9				ret

④ **Unterprogramm-Befehle:** Die Rücksprungadresse liegt auf dem Stapel. Deshalb müssen innerhalb von Unterprogrammen Daten, die mit *push* auf den Stapel gelegt wurden, unbedingt durch *pop* wieder abgehoben werden, damit der Z 80 die Rücksprungadresse findet

Der Befehl *ret* nimmt die obersten beiden Bytes des Stapels ab und lädt sie in den Programmzähler: Es erfolgt der Rücksprung. Der hexadezimale Operationscode für *call NN* lautet »CD . . . .«; für *ret* ist er bekanntlich »C9«.

Zum Abschluß dieses Teils überlegen wir uns, wie ein Maschinenprogramm aussehen muß, das dreimal nebeneinander »ZX-81« auf den Bildschirm schreibt. Dazu nehmen wir das Maschinenlisting zum schnellen Drucken aus Teil 6 und ergänzen es mit Hilfe von Unterprogrammtechniken. *Bild 5* zeigt einen Lösungsvorschlag, der aber nicht im einzelnen besprochen werden soll, weil das Listing teilweise bekannt und außerdem leicht zu verstehen ist. Klaus Herklotz

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	2A	0C	40			ld hl,(400C)
4085	23					inc hl
4086	CD	90	40			call PRINT
4089	CD	90	40			call PRINT
408C	CD	90	40			call PRINT
408F	C9					ret
4090	36	3F			PRINT	ld (hl),3F
4092	23					inc hl
4093	36	3D				ld (hl),3D
4095	23					inc hl
4096	36	16				ld (hl),16
4098	23					inc hl
4099	36	24				ld (hl),24
409B	23					inc hl
409C	36	1D				ld (hl),1D
409E	23					inc hl
409F	23					inc hl
40A0	C9					ret

⑤ **Anwendung eines Unterprogramms:** Dieses Programm sorgt dafür, daß der Text »ZX-81« dreimal auf den Bildschirm geschrieben wird

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

Teil 9: Zugriff auf einzelne Bits

Bit ist die Abkürzung von binary digit (Zweierschritt). Es ist das kleinste Informationselement im Computer und kennt nur die Signalzustände »0« (keine Spannung) und »1« (volle Spannung). Auf diese Bits greifen wir jetzt zu.

Um die Vorgänge im Computer einigermaßen überblicken zu können, haben Mikroprozessor-Entwickler einzelne Bits zu Gruppen zusammengefaßt. Bei der Z-80-CPU sind es 8-Bit-Gruppen, was sich an der 8-Bit-Breite von Speicherzellen, Datenbus und Registern feststellen läßt.

Wenn das c-Register mit der Hex-Zahl 03 geladen wird (Z-80-Assembler: *ld c, 03*), so bedeutet dies, daß im c-Register nur Bit 0 und Bit 1 den Signalzustand »1« aufweisen; Bit 2 bis Bit 7 sind gelöscht *(Bild 1)*.

Die Z-80-Maschinensprache bietet außer dieser direkten Adressierung noch die Möglichkeit, jedes Bit einzeln anzusteuern (bitweise Adressierung). Wie *Bild 2* zeigt, können Bits aus beliebigen Registern wahlweise gesetzt (Signalzustand »1«) oder zurückgesetzt (Signalzustand »0«) werden.

set b, r ordnet dem Bit b des Registers r den Signalzustand »1« zu, während *res b, r* dem Bit b des Registers r den Signalzustand »0« zuordnet.

Die bitweise Adressierung soll wieder praktisch erprobt werden. Versuchen wir dazu, das bc-Registerpaar mit 03 zu laden *(Bild 3)*. Zuerst

C-Register

0	3

0	0	0	0	0	0	1	1
Bit 7 ≙ 2^7	Bit 6 ≙ 2^6	Bit 5 ≙ 2^5	Bit 4 ≙ 2^4	Bit 3 ≙ 2^3	Bit 2 ≙ 2^2	Bit 1 ≙ 2^1	Bit 0 ≙ 2^0

① **Blick ins Detail:** Diese Signalzustände haben die Bits des c-Registers, wenn es mit 03h geladen ist.
$03h = 2^0 + 2^1$, deshalb sind nur Bit 0 und Bit 1 gesetzt

werden auf herkömmliche Art alle Bits des bc-Registerpaares zurückgesetzt (4082h). Danach werden Bit 0 (4085h) und auch Bit 1 des c-Registers gesetzt (4087h). Den Abschluß des Maschinenprogrammes bildet wie gewöhnlich der Rücksprung ins Basic. Nach dem Einpoken und dem Aufruf durch PRINT USR 16514 wird erwartungsgemäß die Zahl 3 auf den Bildschirm geschrieben.

Eigentlich ganz logisch: Logische Verknüpfungen

Ein Programm soll vom Kassettenrecorder in den ZX 81 geladen werden: Man kann den Ladevorgang z. B. nur dann erfolgreich beenden, wenn die Stromversorgung beider Geräte einwandfrei war und wenn die Verbindungskabel richtig angeschlossen wurden.
Beide Aussagen sind durch- und miteinander verknüpft. Man bezeichnet das daher als UND-Verknüpfung (engl: AND): Das Ergebnis fällt nur dann positiv aus, wenn alle Teilaussagen positiv sind. Auf den Mikroprozessor übertragen bedeutet das: Nur wenn alle Eingänge den Signalzustand »1« aufweisen, darf auch der Ausgang den Signalzustand »1« erhalten *(Bild 4)*.

Register r:	a	b	c	d	e	h	l	(hl)
set 0, r	CBC7	CBC0	CBC1	CBC2	CBC3	CBC4	CBC5	CBC6
set 1, r	CBCF	CBC8	CBC9	CBCA	CBCB	CBCC	CBCD	CBCE
set 2, r	CBD7	CBD0	CBD1	CBD2	CBD3	CBD4	CBD5	CBD6
set 3, r	CBDF	CBD8	CBD9	CBDA	CBDB	CBDC	CBDD	CBDE
set 4, r	CBE7	CBE0	CBE1	CBE2	CBE3	CBE4	CBE5	CBE6
set 5, r	CBEF	CBE8	CBE9	CBEA	CBEB	CBEC	CBED	CBEE
set 6, r	CBF7	CBF0	CBF1	CBF2	CBF3	CBF4	CBF5	CBF6
set 7, r	CBFF	CBF8	CBF9	CBFA	CBFB	CBFC	CBFD	CBFE
res 0, r	CB87	CB80	CB81	CB82	CB83	CB84	CB85	CB86
res 1, r	CB8F	CB88	CB89	CB8A	CB8B	CB8C	CB8D	CB8E
res 2, r	CB97	CB90	CB91	CB92	CB93	CB94	CB95	CB96
res 3, r	CB9F	CB98	CB99	CB9A	CB9B	CB9C	CB9D	CB9E
res 4, r	CBA7	CBA0	CBA1	CBA2	CBA3	CBA4	CBA5	CBA6
res 5, r	CBAF	CBA8	CBA9	CBAA	CBAB	CBAC	CBAD	CBAE
res 6, r	CBB7	CBB0	CBB1	CBB2	CBB3	CBB4	CBB5	CBB6
res 7, r	CBBF	CBB8	CBB9	CBBA	CBBB	CBBC	CBBD	CBBE

② **Bitweises Adressieren:** Der Z-80-Befehlsvorrat erlaubt es, Bits einzeln zu setzen oder rückzusetzen (set, res). Jeder der Befehle erfordert zwei 8-Bit-Speicherzellen, da stets CBh vorangesetzt ist

③ **Laden des bc-Registerpaares:** Durch bitweises Adressieren läßt sich z. B. die Zahl 0003h (umständlich) laden

ADRESSE	BYTES			Z-80 ASSEMBLER
4082	01	00	00	ld bc,0000
4085	CB	C1		set 0,c
4087	CB	C9		set 1,c
4089	C9			ret

UND-Verknüpfung	ODER-Verknüpfung	Exklusiv-ODER-Verknüpfung
Symbolschreibweise: E_1, E_2 → A	Symbolschreibweise: E_1, E_2 → A	Symbolschreibweise: E_1, E_2 → A

Funktionstabelle UND-Verknüpfung

Eingänge E_1	Eingänge E_2	Ausgang A
0	0	0
0	1	0
1	0	0
1	1	1

Funktionstabelle ODER-Verknüpfung

Eingänge E_1	Eingänge E_2	Ausgang A
0	0	0
0	1	1
1	0	1
1	1	1

Funktionstabelle Exklusiv-ODER-Verknüpfung

Eingänge E_1	Eingänge E_2	Ausgang A
0	0	0
0	1	1
1	0	1
1	1	0

④ **Logische Verknüpfungen:** Vom Signalzustand an den Eingängen hängt der Signalzustand am Ausgang ab. Die Z-80-CPU simuliert jeweils acht solcher Verknüpfungen auf einmal

Im Gegensatz dazu soll ein Programm erfolgreich ausgeführt werden: Das ist z. B. nur möglich, wenn man die einzelnen Programmzeilen richtig eingetippt hat, oder wenn das Programm richtig geladen wurde.
Die Aussagen sind jetzt durch *oder* miteinander verknüpft. Es liegt eine ODER-Verknüpfung vor (engl.: OR): Das Ergebnis fällt schon positiv aus, sobald nur eine Teilaussage positiv ist. Für unseren Prozessor gilt wieder: Sobald ein Eingang den Signalzustand »1« aufweist, soll auch der Ausgang den Signalzustand »1« erhalten.
Schließlich soll ein Programm durch Drücken der Tasten P oder Q unterbrochen werden. Es wird nur dann unterbrochen, wenn eine der beiden Tasten allein gedrückt wird; ansonsten passiert nichts.

Hier sind die Aussagen durch eine ODER-ähnliche Verknüpfung verbunden. Dieser Fall wird als Exklusiv-ODER-Verknüpfung bezeichnet (engl.: EXOR): Das Ergebnis fällt nur dann positiv aus, wenn die Signalzustände an beiden Eingängen verschieden sind. Für Mikroprozessoren gilt: Sobald beide Eingänge verschiedene Signalzustände aufweisen, erhält der Ausgang den Signalzustand »1«. Was nun bei der Z-80-CPU die jeweiligen Ein- bzw. Ausgänge sind und wie sie verknüpft werden, wird im nächsten Teil verraten. Klaus Herklotz

Klartext im Detail:

Von 8-Bit- und 1-Byte-Befehlen

In Teil 5 der Serie sind sie zum ersten Male aufgetaucht: 8-Bit-, 16-Bit-, 1-Byte- und 2-Byte-Befehle. Da 8 Bit auf jeden Fall 1 Byte ergeben, ist auf Anhieb nicht einzusehen, warum ein und derselbe Befehl in beiden Schreibweisen vorkommt (Schusselei einmal ausgeschlossen). Um es vorwegzunehmen: Dieser kleine Trick hilft, Verwechslungen zu vermeiden.
Eigentlich müßte der Befehl *ld a, N* als 1-Byte-Befehl bezeichnet werden, weil der Akkumulator mit einer Ein-Byte-Zahl geladen wird. Im Prinzip ist das richtig, aber nicht sehr eindeutig. Denn bedenken Sie: Man könnte den Befehl *ld a, N* auch als 2-Byte-Befehl bezeichnen, da zu seiner Ausführung zwei Bytes benötigt werden.
In der gängigen Literatur – an die auch wir uns halten wollen – wird deshalb ein Unterschied gemacht: Soll mitgeteilt werden, daß die Länge des Befehls (also die Gesamtzahl der benötigten Bytes) gemeint ist, so erfolgt die Angabe in Byte bzw. Wort: *ld a, N* ist somit ein 2-Byte- bzw. 2-Wort-Befehl. Will man dagegen bekunden, welche Länge der zu ladende Wert hat, so erfolgt die Angabe in Bit: *Id a, N* ist dann ein 8-Bit-Befehl; *Id bc, N* ein 16-Bit-Befehl. Klaus Herklotz

Softwaretip:

Texteingabe

Ohne viel zu probieren, ermöglicht folgende Eingabemethode für Texte in Verbindung mit PRINT-Anweisungen auf Anhieb zufriedenstellende Ergebnisse. Es gilt lediglich das tatsächliche Zeilenende beim Ausführen der PRINT-Anweisung zu ermitteln, da dieses nicht mit dem Zeilenende beim Schreiben der Anweisung übereinstimmt (siehe FS 11/83, Seite 80).
Um das tatsächliche Zeilenende bereits beim Schreiben der PRINT-Anweisung zu kennen, ist einfach nach den ersten Anführungszeichen ein senkrechter Strich am Bildschirm zu ziehen (Filzstift); er

Vergleichen Sie Basic mit Maschinensprache

Was eigentlich unterscheidet das Programmieren in Maschinensprache von dem in Basic? Machen Sie sich die Unterschiede ruhig einmal klar, um bei Fragen sattelfest Rede und Antwort stehen zu können.
Ein Programm, das in Basic geschrieben ist, wird durch Eintippen von Programmzeilen eingegeben. Ein Maschinenprogramm dagegen wird durch Poken von Speicherzellen eingegeben. Es wurde im Rahmen der Serie »Klartext für den ZX 81« vereinbart, daß es sich dabei um die Adressen der Zeichen einer REM-Zeile am Anfang des Arbeitsspeichers handelt.
Ein Basic-Programm wird durch den RUN-Befehl zum Laufen gebracht, worauf die einzelnen Programmzeilen nacheinander ausgeführt werden. Ein Maschinenprogramm wiederum wird von der Basic-Ebene durch die USR-Funktion abgerufen. Dann werden die einzelnen Speicherzellen hintereinander »erledigt«.
In Basic sind Befehle durch Schlüsselwörter (z. B. PRINT) definiert. Rechnen geschieht mit Variablen (z. B. LET A=3 oder LET B=C). In Maschinensprache sind Befehle durch Befehlscodes (Zahlen) festgelegt. Rechnen erfolgt in Registern (z. B. ld a, 03 oder ld b, c). Beim Rücksprung nach Basic wird beim ZX 81 das bc-Registerpaar ausgedruckt.

markiert das Zeilenende. Bei PRINT-AT-Anweisungen ist zusätzlich die durch AT verursachte Verschiebung zu berücksichtigen. Lautet die Eingabe z.B. 10 PRINT AT 1,10; "XXX..." dann muß der Strich zehn Spalten links vom ersten Anführungszeichen gezogen werden. Nur wenn eine PRINT-Anweisung mit einem Strichpunkt abgeschlossen wird und danach eine PRINT-Anweisung ohne Positionierung erfolgt, dann versagt die Methode. Reinhold Woehler

Klartext für den ZX 81

Teil 10: Logik im Computer

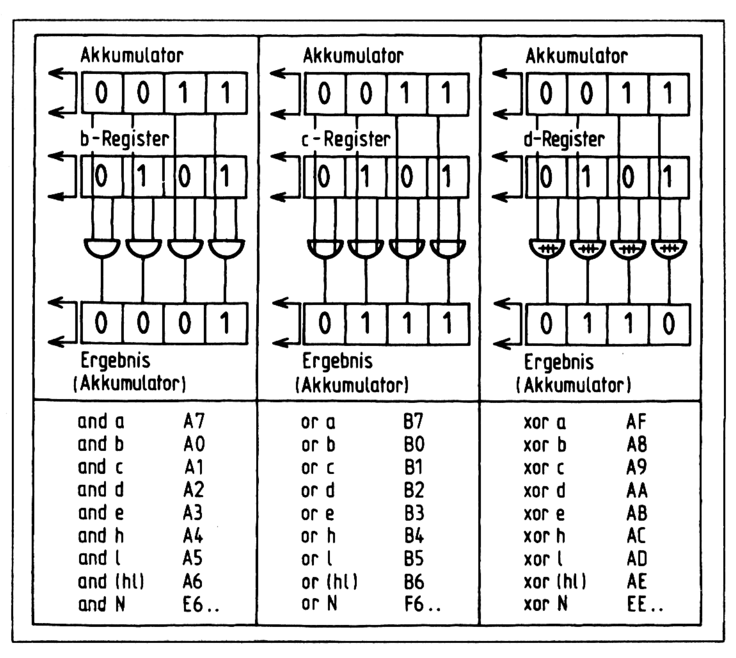
Logische Verknüpfungen sind für den Computer kein Problem, wenn es sich um AND-, OR- oder EXOR-Funktionen handelt. Auch das Addieren und Subtrahieren fußt auf dieser Logik.

Der Z-80-Befehlsvorrat erlaubt es, logische Verknüpfungen zwischen Akkumulator- und Registerinhalten zu programmieren. Wie aus *Bild 1* ersichtlich ist, werden die entsprechenden Bits des Akkumulators paarweise mit den Bits des gewählten Registers verknüpft und das Ergebnis dann im Akkumulator abgelegt.
Der Befehl *and b* z.B. verknüpft die Bits des Akkumulators mit den Bits des b-Registers gleicher Wertigkeit durch ein logisches UND, d.h. das Ergebnisbit wird nur dann gesetzt, wenn das Bit des Akkumulators und das Bit des b-Registers gesetzt ist. Ähnlich verknüpft *or c* die Bits des Akkumulators mit den Bits des c-Registers durch ein logisches ODER, während *xor d* die Bits durch ein logisches Exklusiv-ODER verknüpft.

Logische Beispiele

Die drei Verknüpfungsarten wollen wir sofort erproben *(Bild 2)*. In jedem der drei Programme wird zuerst die Ausgangssituation geschaffen: Der Akkumulator wird mit 03 geladen (Adresse 4082h) und das gewählte Register mit 05 (4084h). Anschließend folgt eine der Verknüpfungsarten (4086h). Zwecks Rücksprung wird letztendlich das Ergebnis der Verknüpfung ins bc-Registerpaar gebracht (4087h bis 4089h). Nach dem Aufruf der Programme mit PRINT USR 16514 wird nach der UND-Verknüpfung »1«, nach der ODER-Verknüpfung »7« und nach der Exklusiv-ODER-Verknüpfung »6« ausgegeben. Wem das nicht klar ist, dem hilft ein Rückblick auf Bild 1: Alle Bits werden paarweise verknüpft!
Ein wenig schleierhaft scheinen die Befehle *or a, and a* und *xor a* zu sein (Bild 1). Denn welchen Sinn kann es haben, den Akkumulator

mit sich selbst zu verknüpfen? Des Rätsels Lösung läßt sich am Beispiel von *xor a* zeigen:
Bei einer Exklusiv-ODER-Verknüpfung wird das Ergebnisbit immer nur dann 1, wenn die Signalzustände beider Teileingänge verschieden sind. Offensichtlich sind die beiden Teileingänge bei *xor a* Bits aus ein und demselben Register (hier: Akkumulator). Das Ergebnisbit ist also unabhängig vom Inhalt des Akkumulators immer 0! Somit ersetzt *xor a* platzsparend den 2-Byte-Befehl *ld a, 00.*



① **Verknüpfungen in der CPU:** Das Ergebnis steht im Akkumulator. Der Akkumulatorinhalt läßt sich auch mit einer 8-Bit-Zahl oder mit dem Inhalt einer durch das hl-Registerpaar beschriebenen Speicherzelle verknüpfen

Arithmetik baut auf Logik

Logische Verknüpfungen ermöglichen dem Computer einfache Rechenoperationen. Das sind im einzelnen Addition und Subtraktion. Dabei werden die einzelnen Bits ziemlich kompliziert verknüpft, was uns aber weiter nicht kümmert.
Wie bei logischen Befehlen sind auch bei arithmetischen Rechenaufgaben der Akkumulator und ein frei wählbares Register die Operanden. So addiert z. B. *add a, b* den Inhalt des b-Registers zum Inhalt des Akkumulators. Ähnlich verhält sich die Angelegenheit bei *sub a, b:* Der Inhalt des b-Registers wird vom Inhalt des Akkumulators subtrahiert. Das Ergebnis der Rechenoperation erhält beide Male der Akkumulator.
Arithmetische Befehle gibt es in 8-Bit- und in 16-Bit-Ausführung *(Bild 3)*. Bei den arithmetischen 16-Bit-Befehlen ist das hl-Registerpaar Ausgangspunkt. Der Befehl *add hl, bc* z. B. addiert das bc- zum hl-Registerpaar.
Versuchen wir jetzt, ein Maschinenprogramm zu schreiben, das ein A in die zehnte Spalte der zehnten Zeile schreibt. Einen Lösungsvorschlag zeigt das Listing aus *Bild 4*.
Wie gewöhnlich erhält das hl-Registerpaar die untere Grenze des Bildspeicherbereichs zugewiesen (Adresse 4082h). Die anzusteuernde Bildspeicherzelle ist die 341ste ihrer Art. Deshalb wird das bc-Registerpaar mit 155h geladen (4085h) und zum Inhalt des hl-Registerpaars addiert (4088h). Letztendlich wird der Buchstabe auf den Bildschirm gebracht (4089h).
8 Bit breite arithmetische und logische Befehle ermöglichen es dem Programmierer, vielseitige Verknüpfungen zwischen Akkumulator und Registern durchzuführen. 16-Bit-Befehle gibt es nur zur Arithmetik. Wie wir später sehen werden, sind die Anwendungsbereiche der arithmetisch/logischen Befehle viel größer, als dies jetzt den Anschein hat.

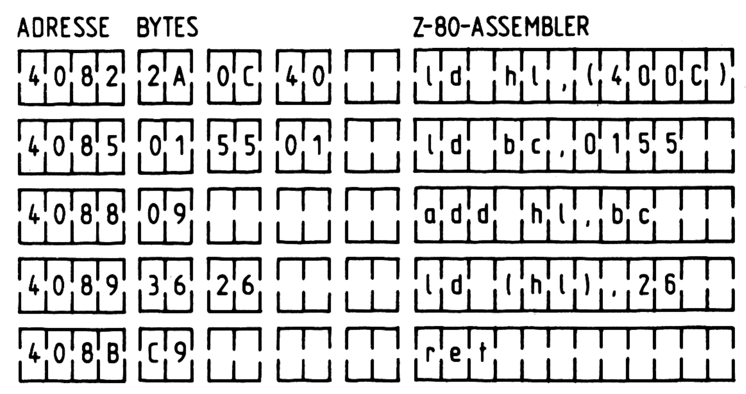
Klaus Herklotz

ADRESSE	BYTES		Z-80-ASSEMBLER
4082	3E	03	ld a,03
4084	06	05	ld b,05
4086	A0		and b
4087	06	00	ld b,00
4089	4F		ld c,a
408A	C9		ret
4082	3E	03	ld a,03
4084	0E	05	ld c,05
4086	B1		or c
4087	06	00	ld b,00
4089	4F		ld c,a
408A	C9		ret
4082	3E	03	ld a,03
4084	16	05	ld d,05
4086	AA		xor d
4087	06	00	ld b,00
4089	4F		ld c,a
408A	C9		ret

② **Logik in der Praxis:** Wer die Ergebnisse verstehen will, dem bleibt das binäre Zahlensystem (Sinclair-Handbuch, Kapitel 24) nicht erspart. Beispiel: 3 xor 5 = 00 000 011 xor 00 000 101 = 00 000 110 = 6h

add a, a	87	sub a, a	97	add hl, bc	09
add a, b	80	sub a, b	90	add hl, de	19
add a, c	81	sub a, c	91	add hl, hl	29
add a, d	82	sub a, d	92	add hl, sp	39
add a, e	83	sub a, e	93		
add a, h	84	sub a, h	94		
add a, l	85	sub a, l	95		
add a, (hl)	86	sub a, (hl)	96		
add a, N	C6	sub a, N	D6		

③ **Arithmetische Befehle:** Beim Addieren von 16-Bit-Zahlen läßt sich auch der Inhalt des Stapelzeigers zum hl-Registerpaar addieren



④ **Bildspeicherzellen ansteuern:** Dieses Programm nutzt den add-Befehl, um eine bestimmte Bildspeicherzelle aufzuspüren. So läßt sich z. B. in die 341ste Bildspeicherzelle (155h) ein A schreiben

Klartext für den ZX 81

Teil 11: Die CPU zeigt Flagge

Um die »Flaggen-Zeichen« des Computers verstehen zu können, machen wir eine kurze Reise in die Vergangenheit.

Ein Segelschiff läuft einen Hafen an, und der Kapitän will den Hafenarbeitern schon von weitem signalisieren, ob sich Ladung auf dem Schiff befindet. Sie haben deshalb vereinbart, daß der Kapitän eine rote Flagge (engl.: Flag) setzt, wenn er *keine* Ladung an Bord hat. Sobald sich Ladung an Bord befindet, ist die Flagge nicht gesetzt. Für einen Hafenarbeiter ist es damit ein leichtes, festzustellen, ob er mit Ladung rechnen muß.

Die Bedeutung des Zero-Flags

Der Z-80-Prozessor hat keine rote Flagge, sondern ein sogenanntes Zero-Flag (Null-Flagge). Das Zero-Flag wird immer dann gesetzt (logisch 1), wenn das Ergebnis einer Operation 0 ist. Dies ist z.B. dann der Fall, wenn der Akkumulator nach Ausführung von *or b* oder *dec a* den Wert 00h aufweist. Fast alle arithmetisch/logischen Befehle beeinflussen das Zero-Flag *(Bild 1)*.
Das Zero-Flag ist eine Grundvoraussetzung für Verzweigungen bzw. bedingte Sprünge in Maschinensprache: Bei *jp z, NN* führt der Prozessor nur dann einen Sprung nach NN aus, wenn das Zero-Flag gesetzt ist. Anderenfalls beachtet der Prozessor die Sprunganweisung nicht und wendet sich der nächsten Adresse zu.

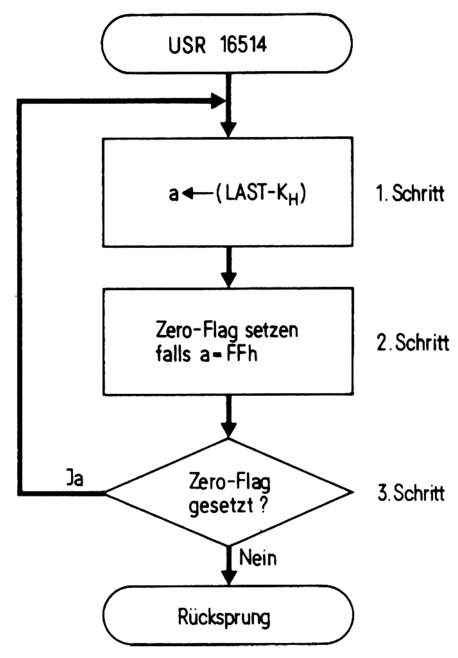
add a, s sub a, s	add a, N sub a, N	and s or s xor s	and N or N xor N	inc s dec s
s ≙ a, b, c, d, e, h, l oder (hl) N ≙ 8-Bit-Zahl				

① **Flag-Beeinflussung:** Sobald das Ergebnis dieser arithmetisch/logischen Operationen 00 ist, wird das Zero-Flag gesetzt

Im Gegensatz dazu führt die CPU bei *jp nz, NN* den Sprung nur dann aus, wenn das Zero-Flag nicht gesetzt ist. *Bild 2* zeigt alle Sprung-Befehle der Z-80-CPU, für deren Ausführung das Zero-Flag verantwortlich ist.

jp z, NN	CA	jp NN wenn Zero-Flag gesetzt
jp nz, NN	C2	jp NN wenn Z-Flag nicht gesetzt
call z, NN	CC	call NN wenn Z-Flag gesetzt
call nz, NN	C4	call NN wenn Z-Flag nicht gesetzt
ret z	C8	ret wenn Z-Flag gesetzt
ret nz	C0	ret wenn Z-Flag nicht gesetzt

② **Bedingte Sprünge:** Fast alle Operationen, bei denen der Programmzähler verändert wird, kann man vom Zustand des Zero-Flags abhängig machen



③ **Flußdiagramm zur Tastenabfrage:** Das Problem ist nach drei Schritten gelöst

Verzweigungen in Maschinensprache

IF-THEN-Verzweigungen, wie wir sie von Basic her kennen, sind prinzipiell auch in Maschinensprache möglich. Versuchen wir z. B. ein Maschinenprogramm zu entwickeln, das solange wartet, bis eine beliebige Taste gedrückt wird, das also die Basic-Zeile ›zz IF INKEY$ = "" THEN GOTO zz‹ ersetzt.
Das Programm muß in etwa der Idee des Flußdiagramms aus *Bild 3* entsprechen. Zuerst muß eines der beiden Bytes der Systemvariablen LAST-K in den Akkumulator gebracht werden (1. Schritt). Beide Bytes von LAST-K weisen den Wert FFh auf, wenn keine Taste gedrückt wird. Eben dann soll auch das Zero-Flag gesetzt werden (2. Schritt). Ist das Zero-Flag gesetzt, dann soll das Programm wieder von vorne ablaufen. Ansonsten erfolgt der Rücksprung ins Basic.
Bild 4 zeigt die Lösung: Zuerst wird der Akkumulator mit dem Inhalt von LAST-K (höherwertiges Byte) geladen (4082h). Vom Akkumulator subtrahieren wir FFh (4085h). Falls vor dieser Operation FFh im Akkumulator stand, so ist jetzt dessen Inhalt 00 und das Zero-Flag ist gesetzt. Damit erfolgt ein Sprung zum Anfang des Programms (4087h).
Ist dagegen eine Taste gedrückt, so wird das Zero-Flag nicht gesetzt. Der Sprung entfällt und die nächste Speicherzelle (408Ah) wird bearbeitet. Nach dem Aufruf des Programmes durch LET Q = USR 16514 erfolgt der Rücksprung ins Basic erst durch Drücken einer beliebigen Taste.
Einen kleinen Nachteil hat diese Methode, das Flag zu setzen, freilich doch: Da es hier mit einem arithmetischen Befehl geschieht, wird der Inhalt des Akkumulators durch *sub a, r* oder *sub a, N* ständig verändert.

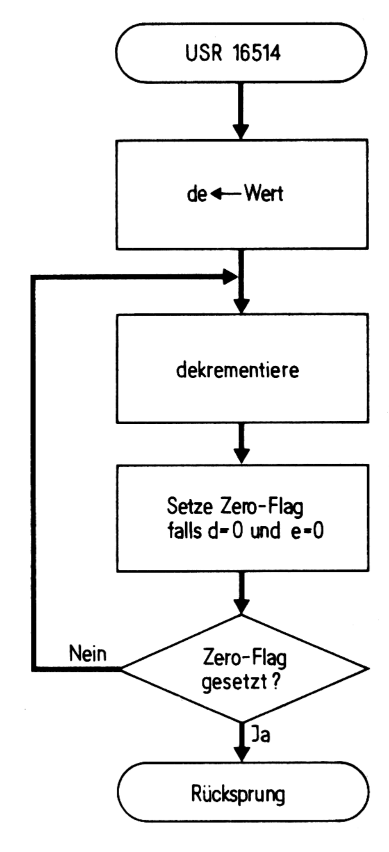
ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	3A	25	40		TASTE	ld a,(4025)
4085	D6	FF				sub a,FF
4087	CA	82	40			jp z,TASTE
408A	C9					ret

④ **Maschinenlisting zur Tastenabfrage:** Der Rücksprung erfolgt erst nach einem beliebigen Tastendruck

⑤ **Vergleiche durch Compare:** *cp r* setzt das Zero-Flag, wenn der Inhalt des Akkumulators und des Registers r gleich ist

cp a	BF	cp h	BC
cp b	B8	cp l	BD
cp c	B9	cp (hl)	BE
cp d	BA	cp N	FE
cp e	BB		

⑥ **Verzögerungsschleife:** Dieses Programm vermindert das Tempo, indem es eine Warteschleife abarbeitet (siehe auch Abb. Seite 70)



Will man das vermeiden, so sollte auf den Compare-Befehl zurückgegriffen werden: *cp r* entspricht praktisch dem Befehl *sub a, r,* nur daß das Ergebnis nicht in den Akkumulator geladen, sondern ausschließlich zum Setzen der Flags verwendet wird *(Bild 5)*. Schreiben Sie zur Übung das Programm aus Bild 4 mit dem Befehl *cp FF*.

Programmier-Probleme werden häufig aus Geschwindigkeitsgründen in Maschinensprache gelöst: Maschinenprogramme sind an Tempo nicht zu überbieten! Manchmal ist dies aber zuviel des Guten und eine Art Bremse muß eingebaut werden.

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	11	00	50			ld de,5000
4085	1B				LOOP	dec de
4086	7A					ld a,d
4087	B3					or e
4088	C2	85	40			jp nz,LOOP
408B	C9					ret

Verzögerungsschleife hilft bremsen

Solche Bremsen sind durchweg Verzögerungsschleifen. Dabei wird der Inhalt eines Register(-paars) in einer Schleife so oft um 1 verringert, bis der Inhalt 0 ist. Wie *Bild 6* zeigt, verwenden wir dafür z.B. das de-Registerpaar und laden es mit einer 16-Bit-Zahl, die für die Dauer der Verzögerung maßgebend ist. An diesem Punkt beginnt die Schleife (engl.: loop): Das de-Registerpaar wird dekrementiert. Dabei ist zu beachten, daß 16-Bit-Inkrementier- und Dekrementier-Befehle keine Wirkung auf Flags haben! Deshalb wird auch das Zero-Flag nicht gesetzt, wenn das de-Registerpaar 0 erreicht.
Wir müssen also einen kleinen Kunstgriff anwenden. Zuerst wird der Akkumulator mit dem d-Register geladen. Danach verknüpft *or e* den Akkumulator mit dem e-Register durch ein logisches ODER: Das Zero-Flag wird genau dann gesetzt, wenn der Akkumulator (mit der Kopie des d-Registers) und das e-Register den Wert 0 haben.
Solange aber das d-Register oder das e-Register ungleich 0 sind, wird das Zero-Flag nicht gesetzt, und es erfolgt ein Sprung zum Schleifenanfang. Nach Aufruf des Programmes durch LET Q=USR 16514 kehrt der Computer erst nach einiger Zeit ins Basic zurück. Finden Sie selbst heraus, welche Verzögerungszeiten sich erreichen lassen. Ein Tip: Die Systemvariable FRAMES ist dabei behilflich.

Zero-Flag-Manipulation bei Einzelbit-Befehlen

In Teil 9 wurde gezeigt, wie durch bitweise Adressierung einzelne Bits eines Registers verändert werden. Nun ist der letzte Bereich der Einzelbit-Befehle an der Reihe:

Register r	a	b	c	d	e	h	l	(hl)
bit 0, r	CB47	CB40	CB41	CB42	CB43	CB44	CB45	CB46
bit 1, r	CB4F	CB48	CB49	CB4A	CB4B	CB4C	CB4D	CB4E
bit 2, r	CB57	CB50	CB51	CB52	CB53	CB54	CB55	CB56
bit 3, r	CB5F	CB58	CB59	CB5A	CB5B	CB5C	CB5D	CB5E
bit 4, r	CB67	CB60	CB61	CB62	CB63	CB64	CB65	CB66
bit 5, r	CB6F	CB68	CB69	CB6A	CB6B	CB6C	CB6D	CB6E
bit 6, r	CB77	CB70	CB71	CB72	CB73	CB74	CB75	CB76
bit 7, r	CB7F	CB78	CB79	CB7A	CB7B	CB7C	CB7D	CB7E

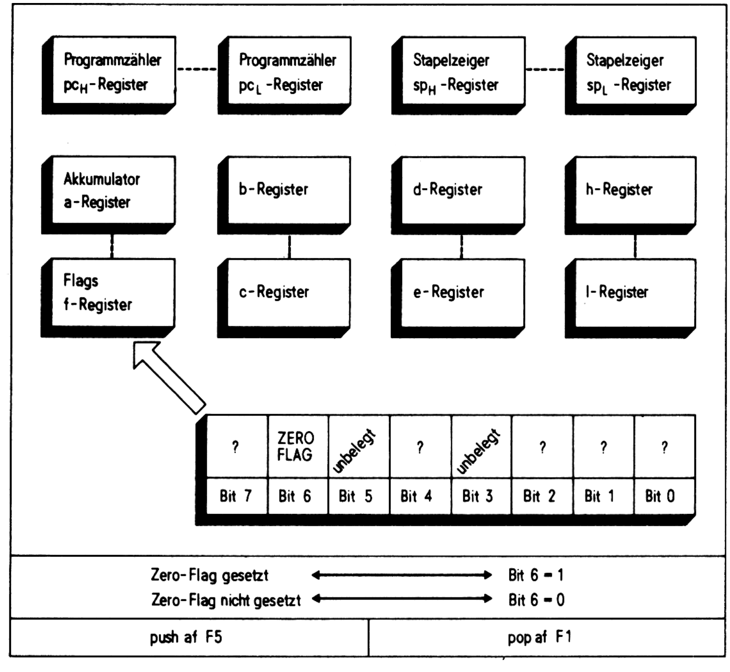
⑦ **Einzelbit-Befehle:** Bits werden auf ihren Wert hin kontrolliert. Das Zero-Flag signalisiert das Ergebnis

Der Befehl *bit b, r* stellt fest, ob Bit b des Registers r gesetzt oder nicht gesetzt ist. Das Ergebnis wird dann wie üblich im Zero-Flag abgelegt. Generell gilt: Das Zero-Flag wird gesetzt, wenn das ausgewählte Bit b des Registers r den Signalzustand 0 aufweist, wenn es also zurückgesetzt ist. Im umgekehrten Fall wird das Zero-Flag nicht gesetzt. Diese Einzelbit-Befehle sind in *Bild 7* aufgelistet.
Bisher wissen wir nur von der Existenz des Zero-Flags und wie man es nutzt. Es ist also an der Zeit, zu erklären, wie das Flag computertechnisch realisiert ist.

Flags stecken im f-Register

Alle Flags (auch die, die wir noch nicht kennen) sind im f-Register enthalten *(Bild 8)*. Jedes Bit im f-Register stellt ein Flag dar: Wenn das Bit den Signalzustand 1 aufweist, ist das entsprechende Flag gesetzt; beim Signalzustand 0 ist das Flag nicht gesetzt. Bit 3 und Bit 5 des f-Registers sind unbenutzt.
Der Akkumulator bildet mit dem f-Register das af-Registerpaar. Die Befehle *push af* und *pop af* erlauben sogar das Speichern dieses Regi-

sterpaares. Mit dem f-Register ist der Z-80-Registersatz im Rahmen dieser Serie fast vollständig. Die restlichen Flags werden in einem der nächsten Teile behandelt. Klaus Herklotz



⑧ **Z-80-Registersatz:** Schlimmer wird's kaum! Jetzt fehlen nur noch die Index-Register. Das Zero-Flag ist Bit 6 des f-Registers

Maschinencode im Griff

Dieses Programm ermöglicht das Eingeben, Betrachten und Verändern von Maschinencode-Programmen. Zunächst fällt auf, daß das Programm hohe Zeilennummern, nämlich solche ab 9000, beansprucht. Das hat den Vorteil, daß man diesen Monitor in den Computer eingeben oder von Cassette laden und dann ein weiteres Programm (welches Maschinensprache-Routinen verwenden soll) wie gewohnt mit niedrigen Zeilennummern eintippen kann. Irgendwo im Programm wird Speicherplatz für den Maschinencode reserviert, üblicherweise durch ein REM-Statement mit entsprechend vielen Bytes hinter REM.

Durch den Befehl RUN 9000 wird der Monitor aufgerufen, der sofort nach der Anfangsadresse des Maschinencodes fragt. Diese ist dezimal einzugeben (beispielsweise 16514). Nun wird hexadezimal der Inhalt der ersten zwölf Bytes ab dieser Adresse angezeigt. Drückt man NEWLINE, so können die nächsten Speicherplätze betrachtet werden. Falls man die gerade angezeigten Bytes verändern möchte, ist einfach der gewünschte neue Inhalt einzutippen. Dies dürfen ein bis zwölf Bytes sein; die Veränderung findet ab der angezeigten Adresse in der erforderlichen Länge statt. Fehlerhafte Eingaben (ungerade Stellenzahl oder für Hex-Code unzulässiges Zeichen) werden ignoriert. Die Eingabe eines »S« (für STOP) bricht das Programm ab; ein »A« bewirkt, daß eine neue Startadresse erfragt wird.

Michael Schramm

```
9000 PRINT AT 19,14;"MONI"
9010 PRINT AT 21,0;"STARTADRESSE
:"
9020 INPUT A
9030 LET A=INT ABS A
9040 SCROLL
9050 PRINT A;TAB 8;
9060 FOR I=0 TO 11
9070 LET M=INT (PEEK (A+I)/16)
9080 PRINT CHR$ (M+28);CHR$ (PEE
K (A+I)-16*M+28);
9090 NEXT I
9100 INPUT D$
9110 IF D$="S" THEN STOP
9120 IF D$="A" THEN GOTO 9010
9130 IF NOT LEN D$ THEN GOTO 924
0
9140 IF LEN D$<>2*INT (LEN D$/2)
 THEN GOTO 9100
9150 FOR I=1 TO LEN D$
9160 IF D$(I)<"0" OR D$(I)>"F" T
HEN GOTO 9100
9170 NEXT I
9180 PRINT AT 21,8;(D$+"
                  ")( TO 24)
9190 FOR I=1 TO LEN D$ STEP 2
9200 POKE A,16*CODE D$(I)+CODE D
$(I+1)-476
9210 LET A=A+1
9220 NEXT I
9230 GOTO 9040
9240 LET A=A+12
9250 GOTO 9040
```

Hex-Monitor: Er hilft beim Schreiben von Maschinencode-Programmen

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

Teil 12: Der relative Sprung

Der Z-80-Mikroprozessor erlaubt zwei Sprungarten: Die relative, die wir hier behandeln werden, und die absolute, die wir bereits kennen.

Bei absoluten Sprüngen wird dem Prozessor die Zieladresse durch zwei Adreß-Bytes ohne viel Geplänkel mitgeteilt. Einen gewaltigen Nachteil hat die Angelegenheit aber doch: Wenn ein Maschinenprogramm verschoben wird – etwa durch ein nachträglich eingefügtes Byte zur Behebung eines Programmierfehlers – dann müssen alle betroffenen Adreß-Bytes nachgestellt werden. Grund genug also, die relative Sprungart anzuwenden.

Relative Sprünge erfordern Geschick

Bei relativen Sprüngen benötigt der Prozessor an Stelle der Zieladresse die relative Sprungweite. Der Programmierer muß sich überlegen, um wie viele Speicherzellen der Sprung vorwärts oder rückwärts erfolgen soll.
Die relativen Sprungbefehle und ihre Codes zeigt *Bild 1.* Dem Operationscode folgt nicht die Zieladresse, sondern die Sprungweite, die durch ein einziges Byte angegeben wird.
Mit relativen Sprüngen spart man deshalb gegenüber absoluten Sprüngen immer ein Byte! Weiterhin können Programmteile verschoben werden, ohne daß Korrekturen notwendig sind: Relative Sprungweiten sind unabhängig von Adressen und demnach überall gültig.
Wie aber wird die Sprungweite angegeben?

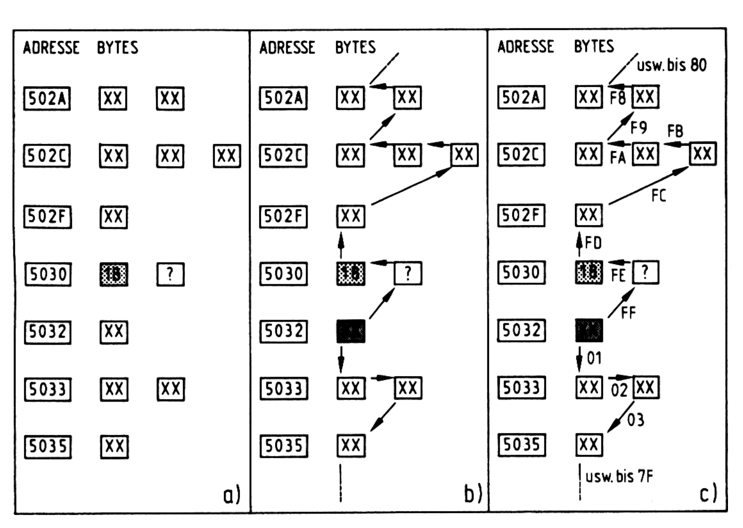
jr E	18	Relativer Sprung um die Entfernung E Anschaulich: jr E ≙ jp pc ± E
jr z, E	28	jr E, wenn Zero-Flag gesetzt
jr nz, E	20	jr E, wenn Zero-Flag nicht gesetzt

① **Relative Sprünge:** Man spart wertvollen Speicherplatz und kann Programmteile verschieben

Das Maschinenprogramm in *Bild 2 a* soll ein x-beliebiges sein. Lediglich Adresse 5030h enthält den Operationscode des relativen Sprungs. Unter der Adresse 5031h ist dann die verschlüsselte Sprungweite einzugeben. Dazu ist die nächste Speicherzelle 5032h markiert (Bild 2 *b*). Sie soll Ausgangspunkt aller folgenden Überlegungen sein, denn auf sie zeigt der Programmzähler nach der Befehlsausführung.

Von Adresse 5032h aus sind alle zu überspringenden Bytes durch Pfeile verbunden und durchnumeriert (Bild 2 *c*): Vorwärts-Sprünge zum Ende des Speichers hin in steigender Reihenfolge ab 01h; Rückwärts-Sprünge zum Anfang des Speichers hin in fallender Reihenfolge ab FFh.

Diese Hex-Zahlen an den Pfeilen geben den Wert an, der jeweils in Adresse 5031h eingesetzt werden muß, um von dort zu der Adresse zu springen, auf die der entsprechende Pfeil deutet: Soll z. B. ein Rückwärts-Sprung zur Adresse 502Ch programmiert werden, so muß Speicherzelle 5031h den Wert FAh erhalten. Bei einem Sprung nach 5035h müßte 03h eingesetzt werden. Daran läßt sich erkennen, daß die Sprungweite bei relativen Sprüngen begrenzt ist.



② **Sprungzielberechnung:** Ein beliebiges Maschinenprogramm (a) hat z. B. unter der Adresse 5030h einen relativen Sprungbefehl. Ausgangspunkt für die Berechnung der Sprungweite ist die Adresse 5032h, weil der Programmzähler vor dem Sprung auf diese Adresse zeigt (b). Bei Vorwärtssprüngen zählt man aufwärts von 01 an, bei Rückwärts-Sprüngen abwärts von FF an (c)

Liegt der Wert der Sprungweite zwischen 01h und 7Fh, so erfolgt ein Sprung nach vorne. Liegt der Wert dagegen zwischen 80h und FFh, so wird eine rückwärtige Zieladresse angepeilt.

Das Tastendruck-Problem wird neu gelöst

In Teil 11 wurde eine Maschinen-Routine behandelt, die so lange wartet, bis eine Taste gedrückt wird. Der darin verwendete absolute Sprung soll nun durch einen relativen ersetzt werden. Die einzige Schwierigkeit dürfte in der Angabe der Sprungweite liegen. Man betrachte deshalb das Listing aus *Bild* 3 und stelle sich wieder die numerierten Pfeile vor! Es wird dann bestimmt klar, daß F9h als Sprungweite gerechtfertigt ist.
Wie auch schon im letzten Teil, erfolgt nach dem Aufruf des Programms mit LET Q = USR 16514 der Rücksprung ins Basic erst durch Drücken einer beliebigen Taste.

So berechnet der Computer die tatsächliche Sprungweite

Soll der Z-80-Mikroprozessor einen relativen Sprung ausführen, so muß er zuerst die Sprungrichtung ermitteln. Die Grenze zwischen den beiden Richtungen liegt wie gezeigt bei 7Fh bzw. 80h. Betrachten wir beide Zahlen im binären Zahlensystem:

7Fh = 0111 1111b und
80h = 1000 0000b

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	3A	25	40		TASTE	ld a,(4025)
4085	FE	FF				cp FF
4087	28	F9				jr z TASTE
4089	C9					ret

③ **Tastendruck:** Ein altes Programm in neuer Auflage wartet, bis eine Taste gedrückt wird

Die Richtung des Sprunges kann praktisch von einem Bit abhängig gemacht werden: Ist Bit 7 nicht gesetzt, dann erfolgt der Sprung nach vorne. Sollte dagegen Bit 7 gesetzt sein, so kündigt das einen Rückwärts-Sprung an.
Bei Vorwärts-Sprüngen ist noch alles klar: Als Sprungweite wird einfach die Anzahl der zu überspringenden Bytes angegeben. Bei Rückwärts-Sprüngen beginnen aber die Schwierigkeiten. Denn wie errechnet der Prozessor aus dem verschlüsselten Wert der Sprungweite die Anzahl der zu überspringenden Bytes?
Des Rätsels Lösung lautet »Zweier-Komplement«. Sollte Bit 7 der verschlüsselten Sprungweite gesetzt sein, dann dreht der Prozessor alle Bits um (Fachsprache: Er invertiert sie oder bildet das Komplement) und zählt 1 dazu. Das Ergebnis liefert die tatsächliche Anzahl der zu überspringenden Bytes. Dazu ein Beispiel mit bekannten Werten:

Sprungweite:	FAh = 1111 1010b
Bits invertiert:	0000 0101b
1 dazu:	(06h) = 0000 0110b

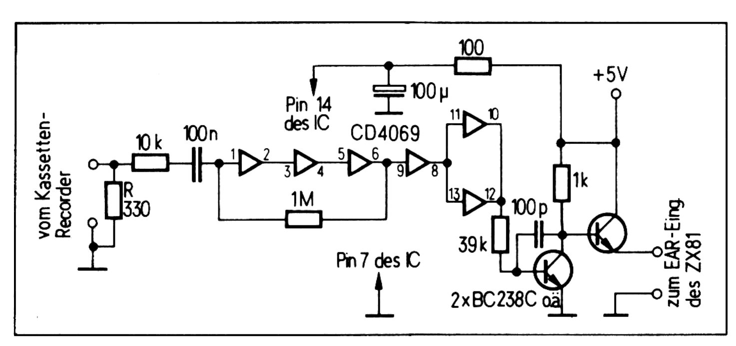
Sollte die verschlüsselte Sprungweite FAh betragen, so muß der Prozessor also 06h Speicherzellen nach hinten »überspringen«. Man überzeuge sich von der Richtigkeit des Ergebnisses durch Nachzählen der Pfeile in Bild 2 c bis zur Zelle FAh. Klaus Herklotz

ZX-81-Hardwaretip:

Signalverbesserung bei LOAD

Das Laden von Programmen, die auf einem anderen Recorder aufgezeichnet worden sind, bereitet oft Probleme. Das liegt meist an einer abweichenden Tonkopf-Einstellung, die einen zu geringen Pegel des wiedergegebenen Signals zur Folge hat. Der Einsatz der hier angegebenen Schaltung führt in derartigen Situationen oft doch noch zum Erfolg. Freilich darf man keine Wunder erwarten; wenn das Signal zu schwach ist, so hilft höchstens die Verwendung eines anderen Kassettenrecorders oder das Verstellen des Tonkopfes.
Die Funktion der Schaltung ist recht einfach: Das Programm-Signal gelangt auf den Eingang eines Sinus-zu-Rechteck-Formers, der mit

dem CMOS-IC CD 4069 realisiert ist *(Bild).* Das Rechteck-Signal erfährt durch die beiden Transistoren noch eine kräftige Stromverstärkung, um den niederohmigen Eingang EAR des ZX 81 ansteuern zu können. Der 100-pF-Kondensator blockiert eingestreute Hochfrequenz und verhindert Eigenschwingungen der Schaltung. Da die Schaltung nur wenig Strom aufnimmt, kann sie ohne weiteres vom 5-V-Spannungsregler des ZX 81 mitversorgt werden.
Anstelle des Lautsprecher/Ohrhörer-Ausgangs kann jetzt auch der DIN-Anschluß eines Recorders das Signal für den ZX 81 liefern; der Wert des Widerstands R muß in diesem Fall entsprechend vergrößert werden (etwa 47 kΩ). Michael Schramm



Begrenzer-Verstärker: Diese kleine Schaltung hilft Signalen auf die Sprünge, damit sie vom ZX 81 akzeptiert werden

Klartext für den ZX 81

Teil 13: Nachbilden von FOR-NEXT-Schleifen

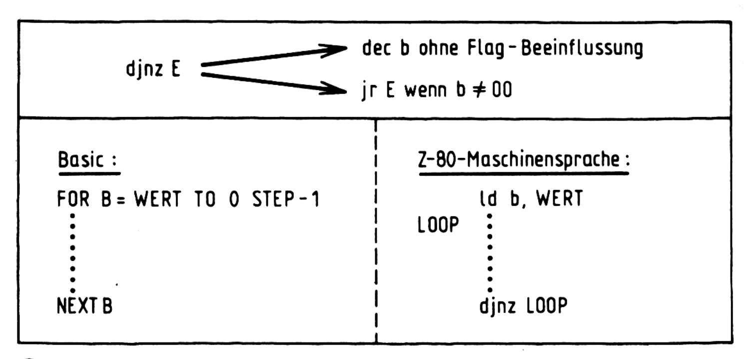
Den einfachen relativen Sprung haben wir im vorangegangenen Teil kennengelernt. Mit einem speziellen Sprungbefehl sind auch FOR-NEXT-Schleifen möglich – nicht so komfortabel wie in Basic – aber immerhin ...

In Teil 12 wurde gezeigt, daß die Z-80-CPU zur Berechnung der Sprungweite eines relativen Sprungs den Inhalt des Akkumulators komplementieren muß. Im Z-80-Befehlsvorrat gibt es dafür zwei arithmetisch/logische Befehle: Der Befehl *cpl* mit dem Code 2F invertiert alle Bits im Akkumulator, bildet also das sogenannte Einer-Komplement. Der Befehl *neg* mit dem Zwei-Byte-Code ED44 invertiert zuerst alle Bits im Akkumulator und zählt dann 1 dazu, was dem Zweier-Komplement gleichkommt (siehe Teil 12). Erproben Sie die beiden Befehle selbständig!

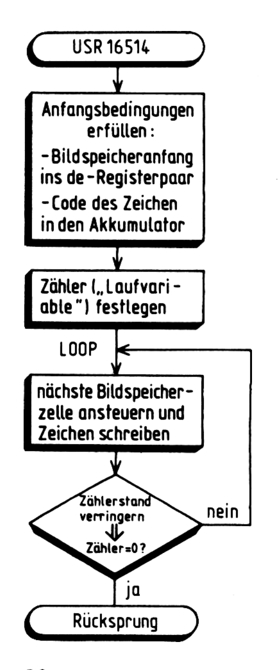
Leistungsstarker Befehl zur Schleifenbildung

Schleifen sind in Basic ein wesentliches Element zur wiederholten Ausführung von Programmteilen. Zur einfachen Schreibweise solcher Wiederholungen bietet Basic dafür die Befehle FOR und NEXT.
In der Z-80-Maschinensprache sind FOR-NEXT-Schleifen ebenfalls realisierbar, jedoch nicht so komfortabel wie in Basic. Als »Laufvariable« kommt nur das b-Register in Frage und die Schrittweite beträgt stets −1.
Zu Beginn jeder Schleife erhält das b-Register einen Wert mit der Anzahl der Durchläufe zugewiesen. Am Schleifenende erscheint dann der Zwei-Byte-Befehl *djnz E* (Abkürzung: decrement and jump if not zero) mit dem Operationscode 10. Er erfüllt gleich zwei Funktionen auf einmal: Zum einen wird der Inhalt des b-Registers um 1 vermindert, ohne dabei Flags zu beeinflussen, zum anderen erfolgt ein relativer Sprung um die Sprungweite E, wenn das b-Register noch nicht auf 0 ist *(Bild 1)*. Sollte es 0 sein – die Schleife ist dann beendet – wird ganz einfach die nächste Speicherzelle bearbeitet.

Die Sprungweite E ist, wie bei relativen Sprüngen üblich, in einem Byte verschlüsselt, das dem Operationscode folgt. Meist liegt die Zieladresse direkt hinter dem Befehl *ld b, WERT* (siehe Bild 1) am Anfang der Schleife (LOOP).



① **Schleifenbefehl:** Der 2-Byte-Befehl djnz E vereint zwei andere Z-80-Befehle. Damit lassen sich FOR-NEXT-Schleifen fast wie in Basic programmieren



② **Schleife in Maschinensprache:** Dieses Programm schreibt die Zeichenfolge »AAAAAA« auf den Bildschirm. Die Zahl der A's wird vom Inhalt des b-Registers bestimmt (Adresse 4088h)

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	ED	5B	0C	40		ld de,(400C)
4086	3E	26				ld a,26
4088	06	06				ld b,06
408A	13				LOOP	inc de
408B	12					ld (de),a
408C	10	FC				djnz LOOP
408E	C9					ret

Die Übersicht behalten mit dem djnz-Befehl

Der djnz-Befehl vereint zwei bekannte Z-80-Befehle. Somit spart man bei seiner Verwendung immer ein Byte, und – das ist noch viel entscheidender – man gewinnt ein hohes Maß an Übersichtlichkeit! Ein Anwendungsbeispiel verdeutlicht das:
Es soll eine Maschinenroutine entworfen werden, die eine vorgegebene Anzahl gleicher Buchstaben (z. B. »AAAA«) auf den Bildschirm schreibt. Betrachten wir dazu gleich das Flußdiagramm und Maschinenlisting *(Bild 2)*.
Zur Abwechslung erhält nicht das hl-, sondern das de-Registerpaar die Adressen der angesteuerten Bildspeicherzellen zugewiesen: Die Systemvariable D-FILE liefert dazu die Adresse der Bildspeichergrenze (Adresse 4082h im Maschinenlisting). Dann wird der Akkumulator mit dem Hex-Code des zu druckenden Zeichens geladen (4086h) und die Anzahl der Schleifendurchläufe ins b-Register gebracht (4088h). Danach wird der Schleifenbeginn in der nächsten Speicherzelle mit LOOP etikettiert.
Im Laufe der Schleife selbst wird der Inhalt des de-Registerpaars erhöht (408Ah) und der Code des gewünschten Zeichens in die durch das de-Registerpaar adressierte Speicherzelle geladen (408Bh). Den Abschluß der Schleife bildet der djnz-Befehl mit der Sprungweitenangabe (408Ch).

Klaus Herklotz

ZX-81-Software:

Malen am Bildschirm

Das nachfolgend beschriebene kurze Programm *(Bild 1)* ermöglicht beliebige Grafiken am Bildschirm. Damit lassen sich z. B. grafisch gestaltete Geburtstagsgrüße, Grundrisse, Irrgärten oder einfach der Phantasie entsprungene Figuren zeichnen. Die »Strichstärke« ist durch den PLOT-Befehl gegeben *(Bild 2)*.
Die Zeilen 20 bis 50 bestimmen den Ausgangszustand (Daten für: Zeichenpunkt blinkend in der linken unteren Ecke des Bildschirms); die Zeilen 50 bis 140 werden ständig durchlaufen. Sie steuern den Zeichenpunkt, bewirken die Ausgabe am Drucker usw. in Abhängigkeit davon, welche Taste gedrückt ist.

Besonders interessant sind die Zeilen 120 und 130, denn hier wird mit logischen Aussagen gerechnet. Viele Basic-Programmierer wissen gar nicht, daß es diese äußerst nützliche Möglichkeit überhaupt gibt. Es werden die X- und Y-Koordinaten des nächsten Punktes in Abhängigkeit von den alten X- und Y-Werten und von A$ (zuletzt gedrückte Taste) bestimmt, wobei die zulässigen Ober- und Untergrenzen für X und Y zu berücksichtigen sind. Selbstverständlich ist das auch durch einige IF-THEN-Anweisungen machbar; kürzer und im Programmlauf schneller geht's auf die hier gewählte Weise (siehe auch Kapitel 10 des ZX-81-Handbuchs).

Der ZX 81 benutzt die Zahlen 0 und 1 für die Ergebnisse FALSCH und WAHR von logischen Aussagen. Die Aussage (A$ = »8« AND X < 63) nimmt also den Wert 1 an, falls die Taste 8 (Cursor) gedrückt und X kleiner als 63 ist. Somit darf X weiter erhöht werden, wenn die rechte Bildkante noch nicht erreicht ist. Entsprechendes gilt für die anderen logischen Aussagen, wobei zu beachten ist, daß der ZX 81 nicht nur 1, sondern jede von 0 abweichende Zahl als logischen Wert WAHR anerkennt; daher kann für X < > 0 einfach X geschrieben werden.

Gestartet wird das Programm durch RUN. Es erscheint sofort in der linken unteren Ecke blinkend der Zeichenpunkt. Die folgenden Tasten sind mit Funktionen belegt.

5: Zeichenpunkt nach links
6: Zeichenpunkt nach unten
7: Zeichenpunkt nach oben
8: Zeichenpunkt nach rechts
0: Lösch-Betriebsart (Zeichenpunkt blinkt)
1: Zeichnen-Betriebsart
N: Zeichenpunkt zum Ausgangspunkt
S: SAVE (Programm und Bildschirminhalt)
C: COPY (Grafik auf Drucker geben)
BREAK unterbricht das Programm.

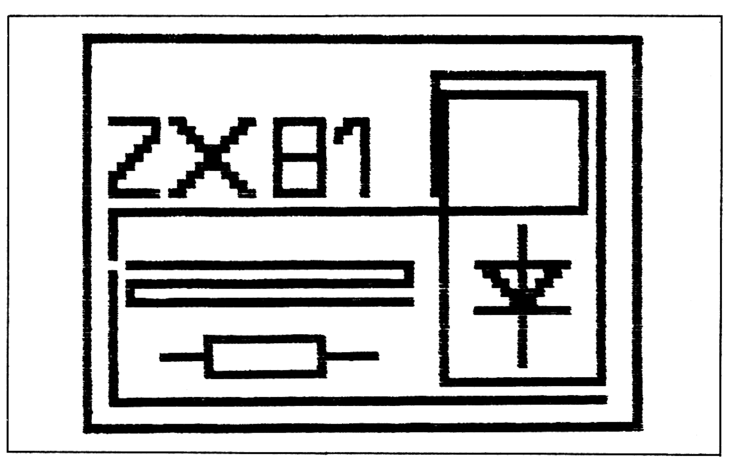
Variablenbelegung:

X: X-Koordinate des aktuellen Punktes (0 bis 63)
Y: Y-Koordinate des aktuellen Punktes (0 bis 43)
M$: Betriebsart
A$: zuletzt gedrückte Taste

Michael Schramm

```
  10 REM GRAPHIK
  20 LET X=0
  30 LET M$="0"
  40 LET Y=X
  50 PLOT X,Y
  60 LET A$=INKEY$
  70 IF A$="C" THEN COPY
  80 IF A$="S" THEN SAVE "GRAPHI
▓"
  90 IF A$="0" OR A$="1" THEN LE
T M$=A$
 100 IF M$="0" THEN UNPLOT X,Y
 110 IF A$="N" THEN RUN
 120 LET X=X+(A$="8" AND X<63)-(
A$="5" AND X)
 130 LET Y=Y+(A$="7" AND Y<43)-(
A$="6" AND Y)
 140 GOTO 50
```

① **Programmlisting »Grafik«:** Die Bewegung des Zeichenpunktes fußt auf logischen Aussagen



② **Grafikbeispiel:** Etwa 10 min waren erforderlich, um dieses Bild mit Hilfe der Cursortasten zu zeichnen

Einführung in Z-80-Maschinensprache:

Klartext für den ZX 81

Teil 14: Schlußpunkte

Dieser speziell für das Sonderheft geschriebene Teil setzt Schlußpunkte hinter die Beschreibung und Demonstration der einzelnen Z-80-Befehle.

Alle Flags befinden sich bekanntlich im f-Register. Eines davon, das Zero- oder Null-Flag, haben wir schon behandelt. Es wird immer dann gesetzt, wenn das Ergebnis einer arithmetischen oder logischen Operation Null ist.

Das Carry-Flag signalisiert einen Übertrag

Das zweite, wichtige Flag nennt sich Carry-Flag (C-Flag) und wird oft auch als Übertrag-Flag bezeichnet. Das Carry-Flag wird immer nur dann gesetzt, wenn eine arithmetische Operation einen Übertrag vom höchstwertigen Bit des Operanden oder des Ergebnisses zur Folge hat. Ein Übertrag vom höchstwertigen Bit des Ergebnisses kann bei einer Addition auftreten. Dies ist z. B. dann der Fall, wenn durch *add a, FF* das Datenbyte FFh zum (hypothetischen) Akkumulatoreninhalt DDh addiert wird (b: Binärzahl):

DDh + FFh = 1101 1101b + 1111 1111b = 1 1101 1100b = 1DCh

Wie aus dem Rechenvorgang ersichtlich ist, paßt das vorderste Bit nicht mehr in den Akku und wandert deshalb ins Carry-Flag! Nachdem diese Operation durchgeführt wurde, steht im Akku die Zahl DCh und das C-Flag ist gesetzt. Sollte einmal kein Übertrag vorliegen, so wird das C-Flag rückgesetzt.
Ein Übertrag vom höchstwertigen Bit des Operanden kann beim Subtrahieren zustandekommen. Dies ist z. B. dann der Fall, wenn durch *sub a, FF* das Datenbyte FFh vom Akkumulatorinhalt DDh subtrahiert wird:

DDh – FFh = 1 1101 1101b – 1111 1111b = 1101 1110b = DEh

Wie aus dem Rechenvorgang ersichtlich ist, ist der Minuend zuerst kleiner als der Subtrahend: Der Prozessor muß sich einen Übertrag „ausleihen“, um die Operation überhaupt durchführen zu können und setzt deshalb das Carry-Flag. Sollte auch hier einmal kein Übertrag erzeugt werden, so wird das Carry-Flag rückgesetzt.
Im Gegensatz zu den arithmetischen Operationen setzen logische Operationen das C-Flag prinzipiell zurück *(Bild 1)*.

Flag-Register

?	ZERO FLAG	unbelegt	?	unbelegt	?	?	CARRY FLAG
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

add a, s sub a, s cp s neg	Setze Carry-Flag (C=1), falls von Bit 7 ein Übertrag erzeugt wird
and s or s xor s	Setze Carry-Flag zurück (C=o)
add hl, ss	Setze Carry-Flag (C=1), falls von Bit 15 ein Übertrag erzeugt wird
scf	Setze Carry-Flag (C=1; Op-code : 37h)
ccf	Komplementiere Carry-Flag ($C=\bar{C}$; Op-code : 3Fh)

① **Carry-Flag:** Dieses Flag kann wahlweise gesetzt, zurückgesetzt, komplementiert oder vom Ergebnis der Operation abhängig gemacht werden

jp c NN	DA	jp NN falls CY = 1
jp nc NN	D2	jp NN falls CY = 0
jr c DIS	38 ..	jr DIS falls CY = 1
jr nc DIS	30 ..	jr DIS falls CY = 0
call c NN	DC	call NN falls CY = 1
call nc NN	D4	call NN falls CY = 0
ret c	D8	ret falls CY = 1
ret nc	D0	ret falls CY = 0

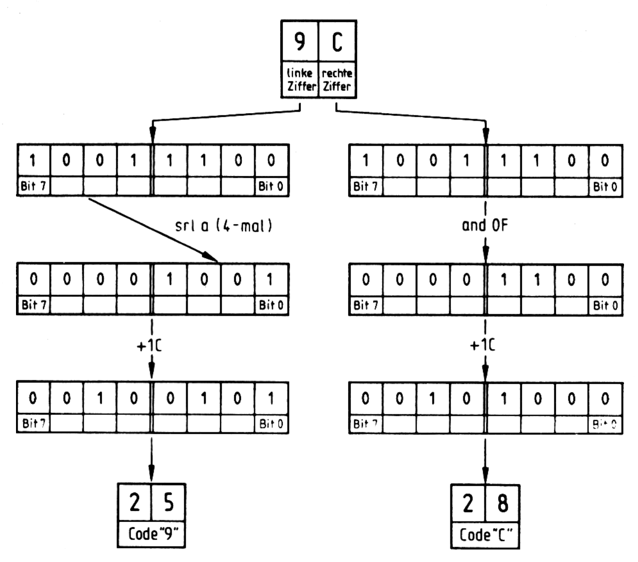
adc a, a	8F	sbc a, a	9F
adc a, b	88	sbc a, b	98
adc a, c	89	sbc a, c	99
adc a, d	8A	sbc a, d	9A
adc a, e	8B	sbc a, e	9B
adc a, h	8C	sbc a, h	9C
adc a, l	8D	sbc a, l	9D
adc a, (hl)	8E	sbc a, (hl)	9E
adc a, N	CE ..	sbc a, N	DE
adc hl, bc	ED 4A	sbc hl, bc	ED 42
adc hl, de	ED 5A	sbc hl, de	ED 52
adc hl, hl	ED 6A	sbc hl, hl	ED 62

② **Wirkung des Carry-Flags:** Sprungbefehle sowie arithmetische Befehle berufen sich auf den Zustand des C-Flags. Sollte Verwechslungsgefahr mit dem c-Register bestehen, so wird es mit CY abgekürzt

In der Z-80-Maschinensprache gibt es eine Vielzahl von Befehlen, für die der Zustand des Carry-Flags von Bedeutung ist.
Da sind zuerst einmal die Sprungbefehle *(Bild 2)*. Bei der Sprungbedingung *c* erfolgt ein Sprung nur dann, wenn das C-Flag gesetzt ist. Umgekehrt erfolgt ein Sprung mit der Bedingung *nc* nur dann, wenn das C-Flag nicht gesetzt ist. Weiterhin gibt es spezielle Arithmetikbefehle, die auf das Übertrag-Flag zurückgreifen.
Man stelle sich z.B. vor, es sollen zwei 32-Bit-Zahlen zusammengezählt werden. Unsere bisherigen 16-Bit-Additionsbefehle reichen gerade für die hintere Hälfte. Nun wissen wir aber, daß diese Befehle einen eventuellen Übertrag ins Carry-Flag bringen! Günstig wäre deshalb ein weiterer Additionsbefehl, der das Carry-Flag berücksichtigt: Sobald bei der letzten Operation ein Übertrag entstanden ist, soll automatisch Eins dazuaddiert werden.
Im Z-80-Befehlssatz sind die beiden Kürzel *adc* (Add with Carry) und *sbc* (Subtract with Carry) zu finden. Der Befehl *adc a, s* entspricht dem Befehl *add a, s*, nur daß das Carry-Flag zusätzlich addiert wird. Genauso wird bei *sbc a, s* das Carry-Flag subtrahiert. Beide Befehle liegen sowohl 8 Bit als auch 16 Bit breit vor (Bild 2).
Wir könnten nun versuchen, ein Maschinenprogramm zu entwickeln, welches zwei beliebige 64 Bit Zahlen zusammenzählt und am Bildschirm als Hexadezimalzahl ausgibt. Mit den bisherigen Mitteln ist dies aber noch unmöglich!

Ausgabe von Zahlen oder Registern

Ein Problem, mit dem früher oder später jeder Programmierer konfrontiert wird, ist das Ausgeben von Registerinhalten. Wie sonst soll eine 64 Bit Zahl ausgegeben werden? Es müßte zumindest gelingen, den Inhalt des Akkus (hexadezimal) auf den Bildschirm zu bringen. So einfach geht dies aber nicht! Denn überlegen Sie einmal:
Jede 8-Bit- bzw. 1-Byte-Hex-Zahl besteht aus zwei Ziffern, von denen wir zunächst den Code ermitteln müssen. Erst danach können die beiden Codes in die entsprechenden Bildspeicherzellen geladen werden. Berechnen wir zunächst die Codes der beiden Ziffern *(Bild 3):* Bei der linken Ziffer müssen alle Bits um vier Stellen nach rechts geschoben werden. Nur so bringt man diese Ziffer allein in ein Register (hier: aus 9Ch wird 09h). Damit der Code der Ziffer vorliegt, braucht bloß noch 1Ch addiert werden (hier: 09h + 1Ch = 25h). Das Ergebnis kann



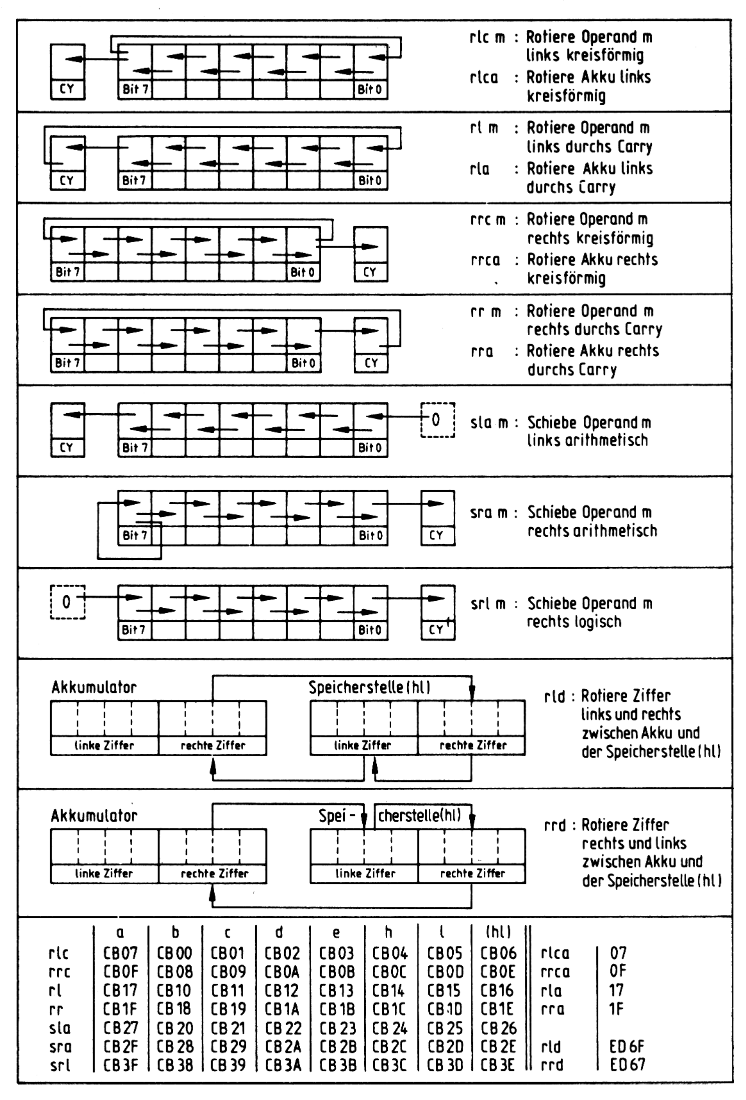
③ **Abspalten von Ziffern:** Nach diesem Schema können die Codes von Ziffern des Akkumulators ermittelt werden

nun in die erste Bildspeicherzelle geladen werden und die linke Ziffer steht am Bildschirm.

Bei der rechten Ziffer verfährt man ähnlich: Zuerst muß die linke Hälfte (auch linkes Nibble genannt) ausgeblendet werden. Dies geschieht mit dem Befehl *and 0F* (Merke: 9Ch and 0Fh = 0Ch; vgl. Teil 10). Danach kann zum Ergebnis wieder 1Ch addiert werden. Der so entstandene Zahlenwert ist der Code der zweiten Ziffer und kann direkt in die zweite Bildspeicherzelle geladen werden.

Rotation ist keine Schiebung

Ein Problem ergibt sich bei der ganzen Angelegenheit: Wie schiebt man am besten die linke Ziffer nach rechts? Z-80-Rotations- und Schiebebefehle sind dafür reichhaltig vorhanden *(Bild 4)*. Im Prinzip werden bei den Schiebebefehlen alle Bits um eine Stelle verschoben. Bei den Rotationsbefehlen findet die Verschiebung dagegen in einem Kreislauf statt. Bemerkenswert ist, daß das Carry-Flag bei den Schiebe- und Rotieroperationen als neuntes Bit verwendet wird.



	a	b	c	d	e	h	l	(hl)		
rlc	CB07	CB00	CB01	CB02	CB03	CB04	CB05	CB06	rlca	07
rrc	CB0F	CB08	CB09	CB0A	CB0B	CB0C	CB0D	CB0E	rrca	0F
rl	CB17	CB10	CB11	CB12	CB13	CB14	CB15	CB16	rla	17
rr	CB1F	CB18	CB19	CB1A	CB1B	CB1C	CB1D	CB1E	rra	1F
sla	CB27	CB20	CB21	CB22	CB23	CB24	CB25	CB26		
sra	CB2F	CB28	CB29	CB2A	CB2B	CB2C	CB2D	CB2E	rld	ED6F
srl	CB3F	CB38	CB39	CB3A	CB3B	CB3C	CB3D	CB3E	rrd	ED67

④ **Rotations- und Schiebebefehle:** Spezielle Befehle verschieben Bits aus Registern. Sogar Ziffern einer Speicherzelle (hl) können abgespaltet werden

Am nützlichsten für unsere Zwecke ist der Befehl *srl a:* Schiebe die Bits des Akkus logisch nach rechts. Dabei wird das höchstwertige Bit auf logisch Null gesetzt! Mit Hilfe dieses Befehls entstand auch das Listing aus *Bild 5.*

In dem Programm wird im ersten Schritt die Adresse der ersten Bildspeicherzelle ins de-Registerpaar gebracht (4082h bis 4086h) und der Akku mit der auszugebenden Zahl geladen (4087h). Um den Akku zu retten, wirft *push af* eine Kopie des af-Registerpaares auf den Stapel. Anschließend werden die Bits des Akkumulators viermal nach rechts geschoben (4084h bis 4091h) und zum Ergebnis 1Ch addiert (4092h). Bei Adresse 4094h wird die linke Ziffer – in Form dieses Ergebnisses – auf den Bildschirm gebracht.

Im zweiten und letzten Schritt erhält das de-Registerpaar die Adresse der zweiten Bildspeicherzelle (4095h), *pop af* stellt die früheren Zustände im Akkumulator wieder her (4096h), und eine logische UND-

ADRESSE	BYTES	LABEL	Z-80-ASSEMBLER
4082	ED 5B 0C 40		ld de,(400C)
4086	13		inc de
4087	3E 9C		ld a,9C
4089	F5		push af
408A	CB 3F		srl a
408C	CB 3F		srl a
408E	CB 3F		srl a
4090	CB 3F		srl a
4092	C6 1C		add a,1C
4094	12		ld (de),a
4095	13		inc de
4096	F1		pop af
4097	E6 0F		and 0F
4099	C6 1C		add a,1C
409B	12		ld (de),a
409C	C9		ret

⑤ **Auf dem Bildschirm schreiben:** Ein Maschinenprogramm bringt den Akkumulatoreninhalt an den Tag

Verknüpfung blendet die linke Ziffer aus (4097h). Abschließend wird auch hier 1Ch addiert und die Ziffer auf den Bildschirm gebracht (409Bh).
Mit Hilfe dieses Programmes lassen sich alle Register und Speicherzelleninhalte innerhalb eines Maschinenprogrammes ausgeben.
Noch ein paar Worte zu den Rotations- und Schiebebefehlen: Im Grunde müßten die Befehle *rlca* (Opcode 07h) und *rlc a* (Opcode CB07h) identisch sein. Dem ist aber nicht so. Beide Befehle beeinflussen die Flags unterschiedlich. So bleibt z. B. das Zero-Flag vom Befehl *rlca* unbeeinflußt, während *rlc a* das Zero-Flag in Abhängigkeit vom Ergebnis setzt bzw. zurücksetzt. Ein weiterer Unterschied ist die Verarbeitungszeit: Der Befehl *rlca* ohne CBh voran wird doppelt so schnell ausgeführt!

Computer als Hex-Rechner

Carry-Flag und Schiebebefehle ermöglichen erstmals die Ausgabe von Hex-Daten. Mit Hilfe dieser Befehle soll nun eine Maschinenroutine erstellt werden, welche zwei 24-Bit-Hexadezimalzahlen addiert und das Ergebnis auf dem Bildschirm ausgibt. Zu allererst stellt sich die Frage, wie beide Summanden eingegeben werden sollen.
Am besten ist es, am Anfang des Programmes zehn Speicherzellen für Daten zu reservieren: Die ersten sechs für die beiden 24-Bit-Summanden, die letzten vier für das 25-Bit-Ergebnis (Übertrag!). Somit können die beiden Summanden direkt am Anfang des Strings mit dem Maschinenprogramm eingegeben werden:

LET A$ = »AAAAAABBBBBB00000000. . .«

legt fest, daß die beiden Zahlen AAAAAAh und BBBBBBh addiert werden, wobei das Ergebnis in die vier weiteren reservierten Speicherzellen geladen wird. Nach Abschluß des Rechenvorgangs bringt das Unterprogramm WRITE das Ergebnis auf den Bildschirm.
Die Bytes des Maschinenlistings aus *Bild 6* sollten wie gewohnt in eine REM-Zeile gepoked werden.

LET Q = USR 16524

bringt das Programm zum Laufen. Nach dem Rücksprung ist der Rechenvorgang beendet und das Ergebnis steht am Bildschirm. Der Programmaufbau im einzelnen:

Am Anfang des Programmes reservieren wie vereinbart *nop's* zehn Speicherzellen (4082h bis 408Bh). Bei Adresse 408Ch erhält das hl-Registerpaar die Adresse des letzten Bytes vom zweiten Summanden. Der Akku wird mit dem letzten Byte des ersten Summanden geladen (408Fh) und dazu das letzte Byte des zweiten Summanden addiert (4092h). Anschließend wird das Ergebnis dieser Operation – das gleichzeitig das letzte Byte des Gesamtergebnisses darstellt – in die vorher reservierte Speicherzelle geladen (4093h).
In den Programmteilen von Adresse 4096h bis 409Dh und 409Eh bis 40A5h wird ein weiteres Byte des Gesamtergebnisses berechnet. Dies geschieht analog zum eben beschriebenen Vorgang. Einziger Unterschied ist der adc-Befehl, der einen Übertrag vom letzten Byte berücksichtigt (409Ah und 40A2h). Der Programmteil von Adresse 40A6h bis 40ABh dient einzig und allein dazu, den Übertrag vom 24sten Bit zu ermitteln und ihn in Form einer Null oder Eins abzuspeichern.
Im weiteren Verlauf des Programmes werden die vier Bytes des Ergebnisses über eine Schleife ausgedruckt (40ACh bis 40BCh). Das Unterprogramm WRITE bringt dazu die Ziffern auf den Bildschirm.

Der ZX 81 wird transparent

Jetzt wollen wir die Vorteile von Maschinensprache voll nutzen: Basic bietet bekanntlich die Möglichkeit, mit Hilfe der PEEK-Funktion den Inhalt beliebiger Speicherzellen auszugeben. Dagegen soll nun ein wesentlich leistungsstärkeres Maschinenprogramm behandelt werden, das gleich einen ganzen Speicherbereich ausgibt.
Geben Sie – ohne viel zu überlegen – die Bytes aus *Bild* 7 mit dem Eingabeprogramm ein. Rufen Sie das Maschinenprogramm anschließend mit dem Kommando
LET Q = USR 16514
auf: 176 Bytes des Speicherbereichs werden ausgegeben. Vorneweg steht in jeder Zeile die Adresse des ersten Bytes, gefolgt von insgesamt acht Bytes. Die Adresse des ersten Bytes auf dem Bildschirm wird, der Einfachheit halber, durch die Systemvariable SEED (zu finden bei Adresse 16434 = 4032h) festgelegt. Durch den RAND-Befehl kann somit von der Basic-Ebene aus die Adresse des ersten Bytes gewählt werden. So bewirkt z. B. die Eingabe von

RAND 16514

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	00	00	00			nop
4085	00	00	00			nop
4088	00	00	00	00		nop
408C	21	87	40		START	ld hl, 4087
408F	3A	84	40			ld a, (4084)
4092	86					add a, (hl)
4093	32	8B	40			ld (408B), a
4096	2B					dec hl
4097	3A	83	40			ld a, (4083)
409A	8E					adc a, (hl)
409B	32	8A	40			ld (408A), a
409E	2B					dec hl
409F	3A	82	40			ld a, (4082)
40A2	8E					adc a, (hl)
40A3	32	89	40			ld (4089), a
40A6	3E	00				ld a, 00
40A8	8F					adc a, a
40A9	32	88	40			ld (4088), a
40AC	ED	5B	0C	40		ld de, (400C)
40B0	13					inc de
40B1	21	88	40			ld hl, 4088
40B4	06	04				ld b, 04
40B6	7E				LOOP	ld a, (hl)
40B7	CD	BE	40			call WRITE
40BA	23					inc hl
40BB	10	F9				djnz LOOP
40BD	C9					ret
40BE	F5				WRITE	push af
40BF	CB	3F				srl a
40C1	CB	3F				srl a
40C3	CB	3F				srl a
40C5	CB	3F				srl a
40C7	C6	1C				add a, 1C
40C9	12					ld (de), a
40CA	13					inc de
40CB	F1					pop af
40CC	E6	0F				and 0F
40CE	C6	1C				add a, 1C
40D0	12					ld (de), a
40D1	13					inc de
40D2	C9					ret

⑥ **ZX 81 als Rechner:** Der Computer addiert die Zahl in den ersten drei Speicherzellen ab 16514 zu der Zahl in den zweiten drei Speicherzellen

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	ED	5B	0C	40		ld de,(400C)
4086	13					inc de
4087	2A	32	40			ld hl,(4032)
408A	06	16				ld b,16
408C	7C				ZEILE	ld a,h
408D	CD	A8	40			call WRITE
4090	7D					ld a,l
4091	CD	A8	40			call WRITE
4094	13					inc de
4095	13					inc de
4096	C5					push bc
4097	06	08				ld b,08
4099	7E				BYTE	ld a,(hl)
409A	CD	A8	40			call WRITE
409D	13					inc de
409E	23					inc hl
409F	10	F8				djnz BYTE
40A1	13					inc de
40A2	13					inc de
40A3	13					inc de
40A4	C1					pop bc
40A5	10	E5				djnz ZEILE
40A7	C9					ret
40A8	F5				WRITE	push af
40A9	CB	3F				srl a
40AB	CB	3F				srl a
40AD	CB	3F				srl a
40AF	CB	3F				srl a
40B1	C6	1C				add a,1C
40B3	12					ld (de),a
40B4	13					inc de
40B5	F1					pop af
40B6	E6	0F				and 0F
40B8	C6	1C				add a,1C
40BA	12					ld (de),a
40BB	13					inc de
40BC	C9					ret

⑦ **Peeken am laufenden Band:** (Bild oben rechts) Eine Maschinenroutine schreibt 176 Speicherzelleninhalte, die durch RAND anwählbar sind, auf den Bildschirm

vor dem Maschinenprogrammaufruf, daß der Speicherbereich ab Adresse 4082h ausgegeben wird. Realisiert ist das Ganze hauptsächlich durch geschickte Schleifenlegung.
Das Registerpaar de erhält zunächst, mit Hilfe der Systemvariablen D-FILE, die Adresse der ersten Bildspeicherzelle (4082h bis 4086h), das hl-Registerpaar die Adresse des ersten auszugebenden Bytes mit Hilfe von SEED (4087h). Nachfolgend wird die erste Schleife festgelegt: Zählregister b erhält den Startwert 16h, was der Anzahl der 22 Zeilen entspricht. In jeder Zeile erscheint zunächst die Adresse: Nachdem der Akku mit dem jeweiligen Byte der Adresse geladen wurde (408Ch und 4090h), erfolgt ein Aufruf der Schreibroutine (408Dh und 4091h). Danach wird, damit etwas Zwischenraum zwischen Adresse und Bytes geschaffen ist, das Bildspeicherzellenregister de zweimal inkrementiert.
Innerhalb dieser Schleife wird eine weitere Schleife verschachtelt. Dazu wird der alte Schleifenzählerstand abgespeichert (4096h) und der neue Startwert 08h (Anzahl der Bytes pro Zeile) festgelegt. Innerhalb dieser Schleife erhält der Akku den Inhalt der auszugebenden Speicherzelle (4099h). Die Schreibroutine bringt diesen Wert auf den Bildschirm (409Ah) und die nächste Bildspeicherzelle sowie das nächste auszugebende Byte werden festgelegt (409Dh und 409Eh). Auch diese innere Schleife endet mit einem djnz-Sprung (409Fh).
Sobald die innere Schleife abgearbeitet wurde, wird das Bildspeicherzellenregister de nochmals um drei erhöht, damit die nächste Adresse auch wirklich am nächsten Zeilenanfang erscheint. *pop bc* holt sich den alten Schleifenzählerstand für die äußere Schleife zurück (40A4h), und die nächste Zeile am Bildschirm kann in Angriff genommen werden. So geht das weiter, bis letztendlich alle Daten ausgeschrieben sind.

Daten speichern

Bei den bisher behandelten Programmen gab es kaum Speicherprobleme: Immer waren genügend Register vorhanden. Sollen aber einmal größere Probleme bewältigt werden, so können sich leicht Zwangssituationen ergeben. An dieser Stelle werden deshalb Probleme, wie sie allgemein beim Speichern auftreten können, aufgedeckt und beseitigt.
Die wohl einfachste Möglichkeit, 16-Bit-Daten zu retten, bieten die

beiden Befehle *push* und *pop: push rp* wirft eine Kopie des Registerpaares rp auf den Stapel, während *pop rp* den obersten 16-Bit-Zahlenwert vom Stapel abnimmt und ihn ins Registerpaar *rp* lädt. Diese Methode, Daten zu speichern, ist zwar einfach, eignet sich aber nur für kurzfristiges Speichern.
Sollte einmal der Speicherplatz sehr knapp werden, d.h. sollten so ziemlich alle Register belegt sein, so können Austauschbefehle weiterhelfen: *ex de, hl* mit dem Operationscode EBh vertauscht den Inhalt der Registerpaare de und hl.
ex (sp), hl mit dem Operationscode E3h wirft eine Kopie des hl-Registerpaares auf den Stapel und nimmt *gleichzeitig* den obersten 16-Bit-Wert vom Stapel ab, vertauscht also die 16-Bit-Daten an der Oberfläche des Stapels mit dem Inhalt des hl-Registerpaares. Der Inhalt des Stapelzeigers bleibt dabei unverändert!
Ernste Schwierigkeiten beginnen, wenn sich Daten, die man vom Stapel abnehmen will, nicht mehr an dessen Oberfläche befinden. Es gibt zwar Befehle, mit denen man im Stapel »herumwühlen« kann *(Bild 8)*, vor deren Verwendung sei aber gewarnt! Es kann allzuleicht passieren, daß der Prozessor dann durch *ret* nicht mehr die Rücksprungadresse erhält, sondern die Daten eines abgespeicherten Registerpaares abhebt. Die Folgen davon müssen wohl nicht geschildert werden.
Die wohl sicherste Möglichkeit, Daten abzuspeichern, bieten 16-Bit-Ladebefehle. Es brauchen bloß zu Beginn des Programmes eine Anzahl von Speicherzellen durch *nop's* freigehalten werden. Im Verlauf des Programmes können 16-Bit-Daten dann durch *ld (NN), rp* eingespeichert bzw. durch *ld rp, (NN)* abgerufen werden.

Mnemonic	Operationscode	symbolische Erläuterung
ld sp, NN	31	sp ← NN
ld sp, (NN)	ED7B	sp_H ← (NN+1); sp_L ← (NN)
ld (NN), sp	ED73	(NN+1) ← sp_H; (NN) ← sp_L
ld sp, hl	F9	sp ← hl
inc sp	33	sp ← sp+1
dec sp	3B	sp ← sp−1
add hl, sp	39	hl ← hl+sp

⑧ **Stapelzeiger als Register:** Das sp-Register ist einem gewöhnlichen 16-Bit-Registerpaar im Datentransport ebenbürtig

Die restlichen Flags

Zero- und Carry-Flag sind die beiden wichtigsten Flags. Wie die Übersicht in *Bild 9* zeigt, gibt es noch vier weitere Flags, die für den Programmierer größtenteils wenig Bedeutung haben. Sie werden deshalb hier weniger ausführlich behandelt.

Flag-Register

SIGN FLAG	ZERO FLAG	unbelegt	HALF CARRY FLAG	unbelegt	P/V-FLAG	ADD/SUB. FLAG	CARRY FLAG
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Befehl	S	Z	H	P/V	N	C	Erklärung
add a,s; adc a,s	!	!	!	!	0	!	8-bit Addition (mit CY)
sub a,s; sbc a,s; cp s; neg	!	!	!	!	1	!	8-bit Subtraktion (mit CY) Vergleich; Zweierkomplement
and s	!	!	1	!	0	0	UND-Verknüpfung
or s; xor s	!	!	0	!	0	0	(Exklusiv-) ODER-Verknüpfung
inc s	!	!	!	!	0	–	8-bit Inkrement
dec s	!	!	!	!	1	–	8-bit Decrement
add hl, ss	–	–	?	–	0	!	16-bit Addition
adc hl, ss	!	!	?	!	0	!	16-bit Addition mit CY
sbc hl, ss	!	!	?	!	1	!	16-bit Subtraktion mit CY
rla; rlca; rra; rrca	–	–	0	–	0	!	Rotation des Akkus
rl m; rlc m; rr m; rrc m sla m; sra m; srl m	!	!	0	!	0	!	Rotation Verschiebung
rld; rrd	!	!	0	!	0	–	Ziffernrotation
cpl	–	–	1	–	1	–	Einerkomplement
scf	–	–	0	–	0	1	Carry wird gesetzt
ccf	–	–	?	–	0	!	Carry-Komplement
bit b,s	?	!	1	?	0	–	Test von Bit b

Zeichenerklärung:

! Flag wird entsprechend beeinflußt
? Flag wird willkürlich beeinflußt
1 Flag wird gesetzt
0 Flag wird zurückgesetzt
– Flag wird nicht beeinflußt

⑨ **Übersicht der Flags:** Nur die aufgelisteten Befehle beeinflussen die Flags

○ Half-carry Flag (Halbübertrag-Flag; H-Flag): Dieses Flag ist dem Carry-Flag sehr ähnlich. Bekanntlich zeigt das Carry-Flag einen Übertrag von Bit 7 an. Das Half-carry Flag zeigt dagegen einen Übertrag von Bit 3 nach Bit 4 an:
H = 1 falls ein Übertrag erzeugt wurde.
H = 0 falls kein Übertrag erzeugt wurde.
Im Z-80-Befehlsvorrat gibt es nicht einmal eine Sprungbedingung, die vom Zustand dieses Flags abhängt. Das H-Flag wird vielmehr vom Prozessor selbst verwendet.

○ Add/Subtract Flag (Subtraktion-Flag; N-Flag): Auch dieses Flag ist für den Programmieralltag bedeutungslos. Das N-Flag zeigt an, ob die letzte Operation eine Subtraktion war:
N = 1 falls die letzte Operation eine Subtraktion war.
N = 0 falls die letzte Operation (vorausgesetzt sie hat das N-Flag beeinflußt) keine Subtraktion war.
Für das N-Flag gibt es ebenfalls keine Sprungbedingungen!

○ Parity or overflow Flag (Parität- oder Überlauf-Flag; P/V-Flag): Das P/V-Flag erfüllt im wesentlichen zwei Aufgaben. Zum einen arbeitet es bei logischen sowie bei Rotations- und Schiebeoperationen als Paritäts-Flag. Es zeigt an, ob in der binären Darstellung des Ergebnisses eine gerade oder ungerade Anzahl von Einsen vorliegt:

P/V = 1 falls die Anzahl der Einsen gerade ist (gerade Parität).
P/V = 0 falls die Anzahl der Einsen ungerade ist (ungerade Parität).

Das P/V-Flag kann zur Kontrolle der Unversehrtheit von Daten eingesetzt werden. Sollte sich z.B. in Folge eines Übertragungsfehlers ein Bit verändert haben, so kann man dies mit Hilfe des P/V-Flags erkennen: Es muß bloß die Parität der Einsen überprüft werden. Die Sprungbefehle zum P/V-Flag lauten:

jp pe, NN	EA	jp NN falls P/V = 1 (gerade Parität; parity even)
jp po, NN	E2	jp NN falls P/V = 0 (ungerade Parität; parity odd)
call pe, NN	EC	call NN falls P/V = 1
call po, NN	E4	call NN falls P/V = 0
ret pe	E8	ret falls P/V = 1
ret po	E0	ret falls P/V = 0

Als weitere Funktion zeigt das P/V-Flag bei arithmetischen Operationen einen Überlauf an. Von den relativen Sprüngen her müßte noch

bekannt sein, daß das höchstwertige Bit des Datenbytes die Sprungrichtung festlegt. Nun ist es in der Zweierkomplementdarstellung einer Zahl üblich, daß das höchstwertige Bit als Vorzeichen verwendet wird: Ist das höchstwertige Bit auf logisch Eins, dann wird die Zahl als negativ behandelt! Wenn auf diese Art gerechnet wird, kann es leicht sein, daß ein Übertrag ins Vorzeichenbit wandert. Für arithmetische Operationen wurde deshalb eine Warneinrichtung konstruiert:

P/V = 1 falls ein solcher Überlauf erzeugt wurde.
P/V = 0 falls kein solcher Überlauf erzeugt wurde.

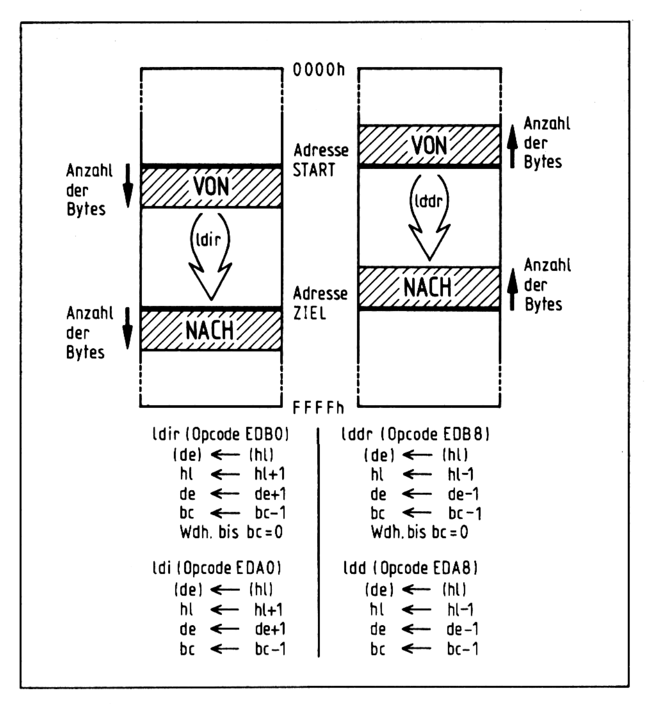
○ Sign Flag (Vorzeichen-Flag; S-Flag): Wie schon beim P/V-Flag erwähnt, stellt das höchstwertige Bit beim Zweierkomplement das Vorzeichen dar. Durch diverse arithmetische, durch logische Operationen und durch Rotations- und Schiebebefehle wird dieses Bit ins S-Flag kopiert:

S = 1 falls das Ergebnis der Operation negativ ist.
S = 0 falls das Ergebnis der Operation positiv ist.
Mit diesem Flag kommt es dann zu folgender Sprungpalette:

jp m, NN	FA	jp NN falls S = 1 (m weil minus)
jp p, NN	F2	jp NN falls S = 0 (p weil positiv)
call m, NN	FC	call NN falls S = 1
call p, NN	F4	call NN falls S = 0
ret m	F8	ret falls S = 1
ret p	F0	ret falls S = 0

Datentransport en masse

Blocktransferbefehle sind so ziemlich das Leistungsstärkste, was der Z-80-Prozessor zu bieten hat. Mit Blocktransferbefehlen können ganze Datenblöcke im RAM-Bereich verschoben werden. Dies bedeutet, daß eine gewünschte Anzahl benachbarter Bytes von einem Teil des Speichers (START) in einen anderen (ZIEL) kopiert werden können. Das Prinzip ist aus *Bild 10* ersichtlich. Bevor der Blocktransfer ausgeführt wird, müssen Registerpaare mit den nötigen Anfangsdaten versehen werden:
Registerpaar bc erhält die Anzahl der Bytes, die kopiert werden sol-



⑩ **Blocktransfer:** Spezielle Befehle kopieren ganze Speicherbereiche

len. Das hl-Registerpaar beinhaltet die Adresse des ersten zu verschiebenden Bytes. Die Zieladresse dieses Bytes erhält das de-Registerpaar.

Abschließend muß man noch entscheiden, ob der Transfer die Bytes mit einer höheren, oder die Bytes mit einer niedrigeren Adresse als die Startadresse übertragen soll. *Idir* überträgt die gewünschte Anzahl von Bytes unterhalb der Startadresse, während *Iddr* die Bytes oberhalb der Startadresse überträgt (Bild 10).

Blocktransfers können als Schleife mit dem 16-Bit-Zählregister bc aufgefaßt werden. Innerhalb dieser Schleife kopiert ein Befehl *ld (de), (hl)* den Inhalt der Speicherzelle, die das hl-Registerpaar festlegt, in die Speicherzelle, die das de-Registerpaar festlegt. Danach werden beim ldir-Befehl die Registerpaare de und hl um Eins erhöht. Der lddr-Befehl erniedrigt diese Registerpaare um Eins. Den Abschluß der

Schleife bilden ein Befehl *dec bc,* der den Schleifenzähler bc um Eins verringert, und ein Sprung zum Anfang der Schleife, solange dieses Zählregister noch nicht Null ist. Die Schleife wird demnach so oft durchlaufen, bis alle Bytes übertragen sind.
Neben den Befehlen *ldir* und *lddr* gibt es noch die beiden Abwandlungen *ldi* und *ldd.* Bei den Abwandlungen fehlt lediglich der Sprung als Abschluß der Schleife, d.h. der Programmierer kann noch einige Bytes in die Transferschleife einfügen. Dabei ist wichtig, daß das P/V-Flag eine Sonderaufgabe erfüllt: Es zeigt an, ob der Schleifenzähler schon Null ist. Solange dies nicht der Fall ist, bleibt auch das P/V-Flag auf logisch Eins. Erst wenn die Schleife abgearbeitet ist, wird es zurückgesetzt. Die Transferschleife muß deshalb mit dem Befehl *jp pe NN* abgeschlossen werden!

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	2A	0C	40			ld hl,(400C)
4085	E5					push hl
4086	01	B5	02			ld bc,02B5
4089	09					add hl,bc
408A	E5					push hl
408B	11	21	00			ld de,0021
408E	19					add hl,de
408F	EB					ex de,hl
4090	E1					pop hl
4091	ED	B8				lddr
4093	E1					pop hl
4094	06	20				ld b,20
4096	23				CLEAR	inc hl
4097	36	00				ld (hl),00
4099	10	FB				djnz CLEAR
409B	C9					ret

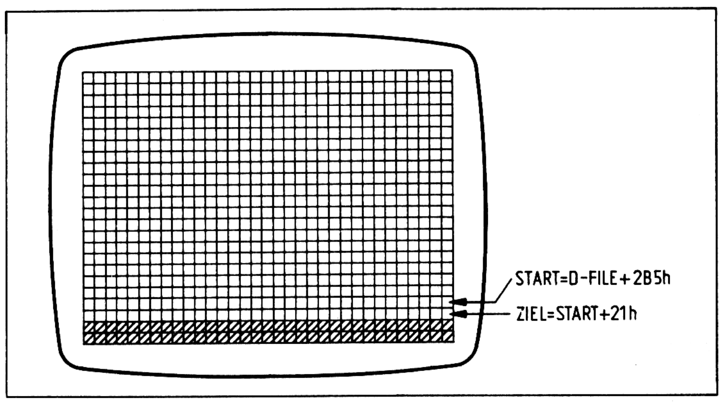
⑪ **SCROLL-Routine:** Ein Maschinenprogramm bewegt den Bildschirminhalt nach unten

Der Bildschirm wird beweglich

Mit den Blocktransferbefehlen lassen sich wie gezeigt Datenblöcke verschieben. Eine Möglichkeit, dies sinnvoll anzuwenden, bietet der Bildspeicher: Im Basic des ZX 81 gibt es den Befehl SCROLL, der den Bildschirminhalt um eine Zeile nach oben bewegt. Es soll nun eine Maschinenroutine vorgeführt werden, die den Bildschirminhalt abwechslungshalber um eine Zeile nach unten bewegt. Die Bytes dieser Routine *(Bild 11)* können wieder in eine REM-Zeile gepoked werden. Jeder SCROLL-Befehl im Basic-Programm kann jetzt durch

LET Q = USR 16514

o. ä. ersetzt werden. Im Gegensatz zur richtigen SCROLL-Routine wird die PRINT-Position nicht verändert! Und so kommt es dazu: Zuerst hilft wie üblich die Systemvariable D-FILE beim Laden des hl-Registerpaares mit der Adresse der Bildspeichergrenze. Dieser Wert ist auch nach dem Transfer noch von Bedeutung und wird deshalb auf den Stapel geworfen (4085h). Das Registerpaar bc erhält die Anzahl der zu verschiebenden Bytes (4086h). Wie aus der Bildschirmskizze von *Bild 12* zu entnehmen ist, stellt dieser Wert 2B5h gleichzeitig den Summanden dar, der zu D-FILE im hl-Registerpaar addiert, die Adresse START ergibt (4089h). Um diesen Wert zu retten, wirft ***push hl*** gleich eine Kopie auf den Stapel (408Ah).



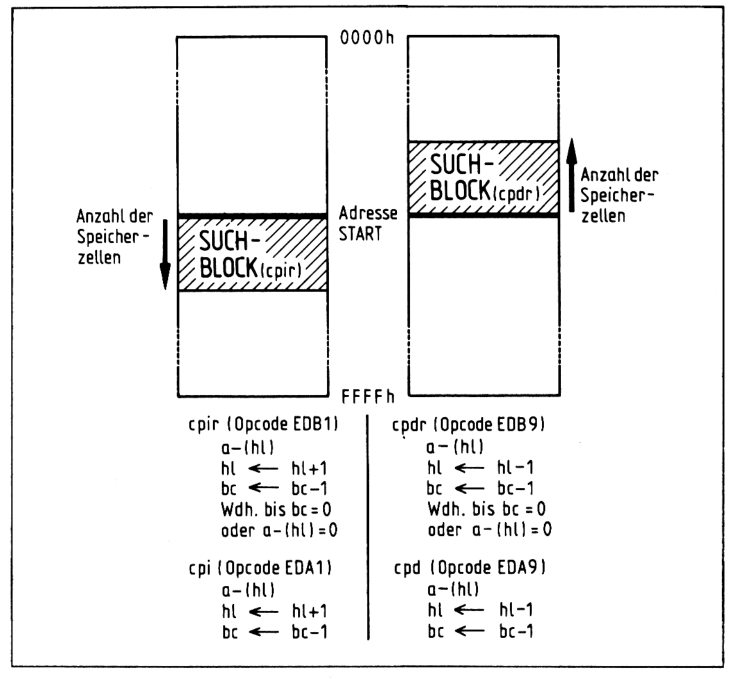
⑫ **Bildschirm:** Rechts unten wird mit dem Transfer begonnen

Registerpaar de muß die Zieladresse erhalten. Dazu braucht bloß 21h zur Startadresse im hl-Registerpaar addiert werden (408Bh bis 408Eh) und das Ergebnis ins de-Registerpaar kopiert werden (408Fh). Abschließend wird die Adresse START wieder ins Registerpaar hl geladen (4090h) und der Transfer *lddr* durchgeführt (4091h). Warum kann hier der Befehl *ldir* nicht verwendet werden?
Der Programmteil nach dem Blocktransfer dient lediglich dazu, die oberste Bildschirmzeile mit Leerzeichen zu überschreiben.
Wer für spezielle Transferzwecke am Bildschirm Maschinenprogramme erstellen will, der kann sich die Arbeit erleichtern: Teil 2 dieser Serie enthält eine detaillierte Skizze des Bildschirms, Teil 3 ein Umwandlungsprogramm für Hexadezimalzahlen: Mit Dezimalzahlen läßt es sich bekanntlich leichter rechnen.

Den Speicherbereich abklappern

Genauso leistungsstark wie die Blocktransferbefehle sind auch die Suchbefehle. Mit Suchbefehlen können ganze Speicherteile nach einem Byte abgesucht werden. Auch hierfür müssen die Register mit den nötigen Anfangsdaten versehen werden.
Registerpaar bc erhält die Anzahl der Speicherzellen, die abgesucht werden sollen. Den hexadezimalen Wert des Suchbytes erhält der Akkumulator, die Adresse der ersten abzusuchenden Speicherzelle wird ins hl-Registerpaar geladen. Abschließend muß man auch hier wieder entscheiden, in welche Richtung die Suche gehen soll: *cpir* sucht unterhalb der Startadresse, während *cpdr* oberhalb der Startadresse sucht *(Bild 13)*.
Zur Arbeitsweise der Suchbefehle: Auch Suchbefehle sind eine Schleife mit dem 16-Bit-Zähler bc. Innerhalb der Schleife vergleicht ein compare-Befehl den Akkuinhalt mit dem Inhalt der Speicherzelle, die das hl-Registerpaar festlegt. Sollten beide Bytes identisch sein, wird das Zero-Flag gesetzt. Nachfolgend erhöht der cpir-Befehl das hl-Registerpaar, während der cpdr-Befehl es um Eins vermindert. Bei beiden Befehlen wird dann der Schleifenzähler bc um Eins reduziert. Solange dieser Schleifenzähler nicht Null wurde, bleibt auch das P/V-Flag gesetzt. Abschließend erfolgt ein Sprung zum Anfang der Schleife solange der Schleifenzähler nicht Null ist *und* das entsprechende Byte nicht gefunden wurde. Wurde dagegen die Schleife verlassen, dann kann man die Ursachen dafür an den Flags ablesen:

Wenn das P/V-Flag zurückgesetzt ist, wurde die Schleife vollständig abgearbeitet. Sollte das Zero-Flag gesetzt sein, so wurde das Byte gefunden: Seine Adresse befindet sich im hl-Registerpaar!
Zusätzlich zu diesen beiden abgeschlossenen Suchbefehlen gibt es wieder offene Abwandlungen: Bei *cpi* und *cpd* fehlt lediglich der Sprung zum Anfang der Schleife. Der Programmierer kann somit wieder einige Bytes einfügen.



⑬ **Suchbefehle:** Ganze Speicherteile werden abgesucht

Dezimalabgleich: Ein feiner Luxus

Wenn nach einer arithmetischen Operation das Ergebnis auf dem Bildschirm steht, beginnt eigentlich erst die Rechnerei: Fein säuberlich muß die ausgegebene Hex-Zahl umgewandelt werden. Da dies sehr zeitraubend ist, wollen wir deshalb zum Abschluß ein Verfahren behandeln, mit dem der Computer dezimale Zahlen ausgibt. Zu den

folgenden Überlegungen ist das 24-Bit-Additionsprogramm aus Bild 6 sehr nützlich. Nur soll diesmal der Rechner nicht mit hexadezimalen Summanden, sondern mit dezimalen Summanden gefüttert werden.

A$ = »006000007000. . .«

veranlaßt den Prozessor, die beiden Zahlen 6000 und 7000 zu addieren. RUN läßt den Rechner die Zahlen einpoken und RAND USR 16524 bringt das Ergebnis D000 auf den Bildschirm.

Das ganze Übel liegt darin, daß der Rechner die beiden eingegebenen Summanden als Hex-Zahlen behandelt. Dem kann aber durch den Befehl *daa* (Decimal Adjust Accumulator: Dezimalabgleich im Akku) mit dem Hex-Code 27 »nachgebeugt« werden! Er muß lediglich nach jeder Addition ins Programm eingefügt werden. Das neue Listing zum Addieren zeigt *Bild 14*. Nach Einpoken und Aufruf wird das gewünschte Ergebnis 13 000 ausgegeben.

ADRESSE	BYTES				LABEL	Z-80-ASSEMBLER
4082	00	00	00			nop
4085	00	00	00			nop
4088	00	00	00	00		nop
408C	21	87	40		START	ld hl,4087
408F	3A	84	40			ld a,(4084)
4092	86					add a,(hl)
4093	27					daa
4094	32	8B	40			ld (408B),a
4097	2B					dec hl
4098	3A	83	40			ld a,(4083)
409B	8E					adc a,(hl)
409C	27					daa
409D	32	8A	40			ld (408A),a
40A0	2B					dec hl
40A1	3A	82	40			ld a,(4082)
40A4	8E					adc a,(hl)
40A5	27					daa
40A6	32	89	40			ld (4089),a
40A9	3E	00				ld a,00
40AB	8F					adc a,a
40AC	32	88	40			ld (4088),a
40AF	ED	5B	0C	40		ld de,(400C)

⑭ **Rechnen mit dem ZX 81:** Der Dezimalabgleich ermöglicht die Addition im Dezimalen (siehe auch Abb. Seite 105)

Über die genaue Funktion des Befehls brauchen wir nicht Bescheid zu wissen. Soviel sei nur am Rande bemerkt: Der Befehl *daa* funktioniert nach einer Addition anders als nach einer Subtraktion. Der Akkuinhalt wird nur dann abgeglichen, sobald eine Ziffer größer als 9h ist. Es wird dann 6h zu dieser Ziffer addiert. Überträge werden entsprechend berücksichtigt.

Die Serie »Klartext für den ZX 81« sollte dem Einsteiger das Programmieren in Maschinensprache verständlich machen. Wer sich noch ausführlicher mit Maschinensprache beschäftigen will, sei auf folgende Bereiche hingewiesen, die nicht von diesem Heft erfaßt wurden: Das sind zum einen die Ein- und Ausgabetechniken, bei denen Interrupts eine maßgebliche Rolle spielen. Zum anderen gibt es noch die beiden Indexregister ix und iy, die eine weitere Adressierungsart ermöglichen. In der FUNKSCHAU werden diese Bereiche im Rahmen der Serie noch abgehandelt.

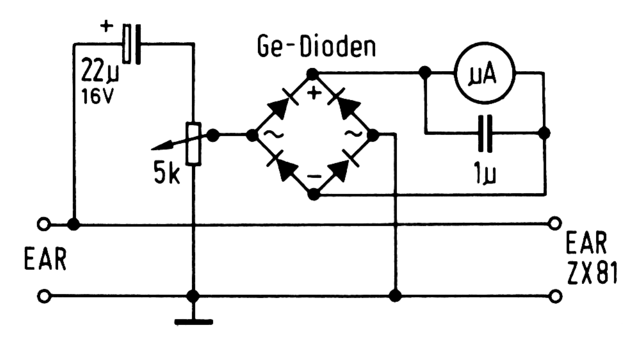
Klaus Herklotz

40B3	13					inc de
40B4	21	88	40			ld hl,4088
40B7	06	04				ld b,04
40B9	7E				LOOP	ld a,(hl)
40BA	CD	C1	40			call WRITE
40BD	23					inc hl
40BE	10	F9				djnz LOOP
40C0	C9					ret
40C1	F5				WRITE	push af
40C2	CB	3F				srl a
40C4	CB	3F				srl a
40C6	CB	3F				srl a
40C8	CB	3F				srl a
40CA	C6	1C				add a,1C
40CC	12					ld (de),a
40CD	13					inc de
40CE	F1					pop af
40CF	E6	0F				and 0F
40D1	C6	1C				add a,1C
40D3	12					ld (de),a
40D4	13					inc de
40D5	C9					ret

ZX-81-Hardwaretip:

Pegelwächter für Ladevorgang

Das Einlesen von Programmen in den ZX 81 ist nicht ganz problemlos, was das Ausfindigmachen des richtigen Lautstärkepegels angeht. Bei Programmen, die mit einem Namen eingelesen werden, geht es noch, aber bei solchen ohne Namen stellt sich der ZX 81 in Punkto Lautstärke recht kleinlich an. Eine einfache Schaltung kann da gute Dienste leisten *(Bild)*.
Hierbei handelt es sich im weitesten Sinne um einen Aussteuerungsmesser: Die niederfrequente Wechselspannung gelangt über den Koppelkondensator und den Einstellwiderstand auf vier Germanium-Dioden, die zu einem Brückengleichrichter geschaltet sind. Hier wird die niederfrequente Wechselspannung gleichgerichtet und an ein Einbauinstrument mit einem Meßbereich von 220 µA und einem Innenwiderstand von 500...600 Ω weitergeleitet. So ein Meßinstrument wird normalerweise in UKW-Tunern verwendet und ist recht preisgünstig zu bekommen. Mit einem Kondensator parallel zum Anzeigeinstrument kann man den Zeiger bremsen.
Und so wird die Schaltung abgeglichen: Man lädt ein Programm ohne Namen in den ZX 81. Ist dies gelungen, läßt man das Programm erneut mit gleicher Lautstärke ablaufen und stellt den Einstellwiderstand so ein, daß der Zeiger des Meßinstrumentes genau auf Mittelausschlag kommt. Nun kann man sehr leicht bei jedem anderen Programm, besonders bei fremdbespielten Kassetten, erkennen, ob die Lautstärke ausreicht oder nicht. Hans-Jürgen Ollech



Ladekontrolle für ZX 81: Die Schaltung hilft beim Laden von Programmen, den richtigen Lautstärkepegel zu finden

Compiler gefällig?

Der auf Kassette angebotene »M-Coder« macht aus einem Basic- ein Maschinencode-Programm.

Der M-Coder ist ein »Compiler« für den ZX 81. Wie der im ROM des Computers untergebrachte »Interpreter« hat auch der Compiler die Aufgabe, der CPU – die ausschließlich ihre eigene Maschinensprache versteht –, alle in Basic erteilten Befehle verständlich zu machen. Der Interpreter löst die Aufgabe, indem er nach dem Starten des Basic-Programms gemütlich immer nur den Befehl übersetzt, der vom Programmablauf her an der Reihe ist. Taucht der gleiche Befehl danach wieder auf, so muß er jedesmal neu übersetzt werden. Das macht Interpreter ziemlich langsam.
Compiler übersetzen ein Basic-Programm nicht schlückchenweise, sondern machen das vor dem eigentlichen Programmstart auf einem Zug. Normalerweise ist der Compiler zur anschließenden Programmausführung nicht mehr erforderlich; er kann gelöscht werden (nicht beim M-Coder!), ebenso wie das Basic-Programm. Übrig bleibt der Maschinencode des Basic-Programms, den die CPU unmittelbar, d.h. schnell ausführt.

Ganze Zahlen sind Pflicht

Ins Haus kommt der M-Coder als rd. 3 KByte umfassendes Programm auf einer Compactkassette. Das Laden dauert knapp 2 min (davon nimmt ein Ladetest ca. 20 s in Anspruch), worauf sich der M-Coder selbst »zur Stelle« meldet. Jetzt kann entweder das zu übersetzende Basic-Programm wie üblich von Band geladen werden, oder man tippt es direkt ein.
Beim Eintippen kostet es etwas Überwindung, der Bedienungsanleitung Folge zu leisten. Dort heißt es: »Programm beginnend mit Zeilennummer 5 eingeben«. Nun sind aber Zeile 5 und die folgenden bereits vom M-Coder benutzt. Tatsächlich dürfen jedoch ab Zeile 5 alle Zeilen bedenkenlos überschrieben werden.
Leider zwingt der Compiler dem einzugebenden Basic-Programm einige drastische Einschränkungen auf, so daß nicht jedes beliebige Ba-

sic-Programm vom M-Coder übersetzt werden kann. Die wichtigsten Einschränkungen sind:

- Es dürfen nur ganze Zahlen (max. ± 32767) verwendet werden.
- Von den mathematischen Funktionen des ZX 81 sind nur die vier Grundrechenarten erlaubt.
- Mit DIM darf nur ein einziges (eindimensionales) Zahlenfeld definiert werden.
- RND erzeugt eine Zufallszahl zwischen 0 und 32767.
- Kommas nach PRINT-Befehlen haben für den M-Coder die Bedeutung von Strichpunkten.
- FOR-NEXT-Schleifen müssen mit STEP = 1 laufen.
- String- und Boolesche Operationen sind nicht erlaubt.

Der M-Coder wird deshalb nur in den seltensten Fällen ein Basic-Programm auf Anhieb übersetzen können. Normalerweise muß das Basic-Programm dem Compiler angepaßt werden, was im vollen Umfang wiederum oft nicht möglich ist. Hier hilft der STOP-Befehl weiter, denn der M-Coder übersetzt ein Basic-Programm nur bis zu der Stelle, wo ein STOP-Befehl steht. Damit läßt sich also auch nur der Anfang eines Basic-Programms übersetzen, wenn sich dessen Rest einfach nicht anpassen lassen will.

Eingetippte oder geladene Programme können wie üblich getestet werden, bis sie wunschgemäß laufen. Dann wird's spannend, weil jetzt durch Starten eines Maschinenprogramms der Compiler erstmals in Aktion tritt. Wurden alle Einschränkungen beachtet, läßt der M-Coder das eingegebene Basic-Programm am Bildschirm dreimal Revue passieren – und fertig ist die Übersetzung.

Findet der M-Coder jedoch eine unzulässige Eingabe, so macht er wie der Basic-Interpreter mit einem inversen »S« auf die Fehlerstelle aufmerksam.

Hat die Übersetzung geklappt, kann das übersetzte Basic-Programm (bzw. der entsprechende Teil) durch Eintippen der jeweiligen Zeilennummern wie üblich gelöscht werden. Vom Maschinenprogramm, das der Compiler in einer versteckten REM-Zeile untergebracht hat, ist allerdings nichts zu sehen. Aufgerufen mit LET L = USR 18823 führt es aber die gestellte Aufgabe in atemberaubendem Tempo aus, während das Bild, wie im SLOW-Modus, ständig aufgebaut bleibt.

Zum Tempo ein Beispiel: Die Anweisung LET A = 10/5 + 10/6 – 2 von einer FOR-NEXT-Schleife 1000mal ausführen zu lassen dauert in Basic 55 s (SLOW-Modus) bzw. 13 s (FAST), in Maschinensprache dagegen nur 2,6 s.

Ballast beim Speichern

Läuft alles zur Zufriedenheit, kann man ans Abspeichern denken. Das erfordert wie üblich den SAVE-Befehl und dauert auch bei sehr kurzen Maschinencode-Programmen gut 1,5 min.
Des Rätsels Lösung zeigt sich nach dem Wiedereinlesen des Programms – dann nämlich meldet sich prompt der Compiler zu Wort, der zwangsläufig beim M-Coder mit dem eigentlichen Maschinenprogramm abgespeichert wird, weil das Maschinenprogramm nicht ohne den Compiler auskommt. Der im übrigen voll funktionsfähige Compiler ist also immer mit dabei, selbst wenn man ihn nicht haben möchte.
Trotz der drastischen Einschränkungen macht es Spaß, mit dem M-Coder zu arbeiten und vor allem mit ihm zu experimentieren. Die Anschaffung ist er allemal wert (Preis zum Redaktionsschluß 30 DM; Profisoft GmbH, Osnabrück), wenn man die Vorteile der Maschinenprogrammierung teilweise nutzen möchte, ohne irgendwelche Kenntnisse der Z-80-Programmierung zu haben. -ll

Tipsammlung zum M-Coder

Mit dem auf Seite 40 vorgestellten M-Coder ist beim Einhalten bestimmter Regeln ein beträchtliches Steigern der Arbeitsgeschwindigkeit des ZX 81 möglich. Das ist insbesondere bei bewegten Grafiken nützlich.
Nachstehende Maßnahmen helfen, bestehende Basic-Programme so zu verändern, daß sie vom M-Coder abzeptiert werden (siehe auch *Tabelle*).
○ FOR-NEXT-Schleifen: Der M-Coder duldet nur FOR-NEXT-Schleifen mit der Schrittweite +1. Bei negativer Schrittweite muß deshalb die Schleife durch einen Sprungbefehl abgearbeitet werden. Dazu ein Beispiel: Das ursprüngliche Basic-Programm

```
10 LET H=5
15 FOR V=16 TO 3 STEP -1
20 PRINT AT V,H;"X";AT V+1,H;" "
25 NEXT V
```

wird verändert zu

```
10 LET H=5
15 LET V=16
20 LET W=V+1
25 PRINT AT V,H;"X";AT W,H;" "
30 LET V=V-1
35 IF V> =3 THEN GOTO 20
```

○ RND: Der M-Coder erzeugt Zufallszahlen zwischen 0 und 32767. Um z. B. für Bildschirmgrafiken Zufallszahlen zwischen 0 und 20 zu bekommen, ist die ursprüngliche Basic-Anweisung

```
10 LET C=INT (20*RND)
```

zu ändern in

```
10 LET C=INT (RND/(32767/20))
```

○ INPUT: Das Basic-Programm muß so geändert werden, daß nach INPUT nur ein einzelner Buchstabe stehen darf, z. B. INPUT A. Die Anweisungen INPUT A1, INPUT ANTON, INPUT A$ werden nicht akzeptiert.

○ STOP/BREAK: Oft ist beim M-Coder das Anhalten von Programmen durch BREAK nicht möglich. In das zu compilierende Programm sollte deshalb an Schlüsselstellen folgende Notbremse eingebaut werden:
IF CODE INKEY$=118 THEN GOTO nn. Das Drücken von NEWLINE unterbricht dann das Programm. Unter nn ist die Zeilennummer der STOP-Anweisung einzugeben, die am Programmende steht. Die Anweisung . . .THEN STOP ist nicht erlaubt, da STOP nur einmal am Programmende vorkommen darf.

○ PRINT: Strings dürfen eine maximale Länge von 127 Zeichen nicht überschreiten, da sonst das System abstürzt. Längere Strings sind auf mehrere PRINT-Anweisungen aufzuteilen.
Ein in der 22. Zeile ausgegebener String darf nur 31 Zeichen lang sein; die letzte Stelle muß frei bleiben.
PRINT AT erfordert eindeutige Koordinaten, z. B. PRINT AT A,B;"XYZ" oder PRINT AT 3,5;"XYZ". Nicht compiliert wird: PRINT AT A,B+3;"XYZ". B+3 muß durch eine neue Variable ersetzt werden.

○ GOTO, GOSUB, RUN: Die Zielzeilen müssen im Programmlisting tatsächlich vorhanden sein.

○ Speicherbedarf der compilierten Programme: Im allgemeinen benötigen die in Zeile 0 und 2 verbleibenden Maschinencodeteile min. 2600 Byte Speicherplatz. Wiederholtes Editieren im Basicteil und Compilieren läßt den Speicherbedarf rasch anwachsen. Um Platz zu sparen, sollte ein bereits compilierfähiges Basic-Programm zwischendurch ohne Zeile 0 und 2 auf Band gespeichert, anschließend neu in den Rechner geladen und endgültig compiliert werden.
Der letztlich verbleibende Basicteil sollte außer Zeile 0 und 2 nur mehr aus dem Aufruf der Maschinencoderoutinen und der Zuweisung der Variablenwerte für A. . .Y bestehen. Karl Heinz Krehan

Tabelle: Unterschiede zwischen M-Coder und ZX-81-Interpreter

Funktion:	ZX-81-Interpreter:	M-Coder:
INKEY$	If INKEY$ = "G" THEN. . .	IF CODE INKEY$ = 44 THEN. . .
PRINT AT	PRINT AT A,B + 3;"U"	LET C = B + 3 PRINT AT A,C;"U"
PRINT »String«	PRINT "ABCDEFGH. . ." (Länge beliebig)	PRINT "ABCDEFGH. . ." (Länge max. 127 Zeichen)
GOTO	GOTO 25 ist erlaubt, wenn nur Zeile 30 vorh.	GOTO 30
RND	LET C = INT (26 * RND) + 1 (Zufallszahl 0. . .26)	LET C = INT(RND/1250) (Zufallszahl 0. . .26)
Grafik	24 Zeilen mit POKE 16418,0 22 Zeilen voll nutzbar	nicht erlaubt! 22. Zeile nur 31 Zeichen!
STOP	IF. . .THEN STOP	nicht erlaubt! STOP nur am Programmende!
Stringvariable A$. . .Z$	LET A$ = "ABCDEFG. . ."	nicht erlaubt
SGN	LET F = SGN(W)	nicht erlaubt
Variable	LET A1 = 246 LET ANTON = 246	nur LET A = 246 möglich! (Buchstaben A. . .Y)
INPUT	INPUT A$. . .Z$ INPUT A1 INPUT ANTON	nicht erlaubt! nur INPUT A. . .Y möglich!
BREAK	immer wirksam	fallweise unwirksam!

Noch einmal M-Coder:

Programmlisting ohne Gewalt

Das Programm M-Coder hat unter anderem eine sehr lästige »Makke«: Nach jeder Eingabe einer Programmzeile springt es in die nichtssagende erste Zeile des Programms (0 REM ZXGT). Das zur Umwandlung eingegebene Programm, an dessen Listing man eigentlich interessiert ist, ist nur mit Gewalt, z.B. mit LIST 30, auf den Bildschirm zu bekommen und verschwindet bei Betätigung der Cursortasten oder einer anderen Eingabe sofort wieder zugunsten der Zeile 0. Man ist also praktisch gezwungen, blind zu arbeiten.
Beseitigen kann man dies, indem man folgende Befehle eingibt:
POKE 16518,0
POKE 16519,0
LIST 30
Danach ist der Effekt verschwunden, allerdings darf man keinesfalls das LIST 30 vergessen, und man sollte auch nicht die (ohnehin uninteressanten) Zeilen 0 und 2 listen lassen, da sonst dasselbe Problem mit Zeile 2 auftreten kann. Ab Zeile 3 ist der M-Coder nun problemlos zu listen; auch die Cursortasten funktionieren wieder. Wolf-Dieter Roth

Software:

Elektronischer Notizblock

Ist es schon wieder Zeit für einen Ölwechsel, oder wurden beim letzten Service die Zündkerzen gewechselt? Die Daten zum Beantworten solcher oder ähnlicher Fragen kann man sich notieren – oder elektronisch speichern. Dazu ein Programm für den ZX-81 mit 16-KByte-RAM. Der elektronische Notizblock bietet 15 Bildschirmseiten mit je 20 Zeilen à 32 Zeichen.
Die Seiten lassen sich gezielt aufrufen, das automatische Suchen nach einem bestimmten Begriff ist jedoch nicht möglich. Außerdem sollte man sich darüber klar sein, daß für jeden Blick in den Notizblock das Laden von Kassette nötig ist (Ladedauer des Programms selbst etwa 4 min.).
Nachdem das Programm *(Bild 1)* eingetippt ist, wird es mit RUN gestartet. Nach dem Einspeichern von Daten darf es dagegen nur noch

mit GOTO 10 gestartet werden, sonst gehen die Informationen verloren. Von Kassette aus erfolgt Autostart. Dann erscheint ein Inhaltsverzeichnis, das die jeweils erste Zeile der 15 Seiten (bei leeren Seiten Punkte) darstellt *(Bild 2)*.

Durch Tippen von »16« gelangt man in den Schreibmodus. Der Computer fragt nach der Seitenzahl und zeigt nach der Eingabe die erste Seitenzeile. Da die erste Zeile einer jeden Seite automatisch ins Inhaltsverzeichnis übernommen wird, ist es sinnvoll, alle zu numerieren (von 1 bis 15) und einen Kurztitel einzutippen.

Bei jedem Eingabemodus wird durch »X« das Inhaltsverzeichnis aufgerufen (gilt auch für das Korrigieren und Abspeichern auf Kassette). Wird im Schreibmodus »Y« gedrückt, kann eine neue Seite begonnen werden. So lassen sich alle 15 Seiten vollschreiben.

```
   1 REM NACH ERSTEM RUN NUR NOC
H MIT GOTO 10 STARTEN
   2 DIM A$(15,20,32)
   3 FOR D=1 TO 15
   5 LET A$(D,1)="..............
................"
   7 NEXT D
  10 PRINT "INHALTSVERZEICHNIS<>
SEITE NR:"
  15 PRINT "====================
============="
  20 FOR D=1 TO 15
  25 IF A$(D,1)="
                 " THEN LET A$(D
,1)=".............................
...."
  30 PRINT A$(D,1)
  40 NEXT D
  50 PRINT "████████████████████
███████████"
  70 PRINT "16)DATEI SCHREIBEN"
  80 PRINT "17) ""     KORRIGIERE
N"
  85 PRINT "18) ""     SPEICHERN
AUF CASSETTE"
  90 PRINT "MIT >X<  INHALTSVERZ
EICHNIS"
  95 LET F$="...................
..............."
 100 GOSUB 700
 120 LET A=VAL C$
 205 CLS
 206 IF A<1 OR A>18 THEN GOTO 10
 207 IF A=16 THEN GOTO 500
 208 IF A=17 THEN GOTO 300
 209 IF A=18 THEN GOTO 430
 210 FOR N=1 TO 20
 215 IF A$(A,N)="...............
..................." THEN LET A$(A
,N)="
     "
 220 PRINT A$(A,N)
 230 NEXT N
 231 INPUT C$
 238 IF C$<>"X" THEN GOTO 231
 240 IF C$="X" THEN CLS
 245 IF C$="X" THEN GOTO 10
 300 PRINT "SEITENNUMMER?"
 310 GOSUB 700
 315 CLS
 316 LET A=VAL C$
 317 IF A>15 OR A<1 THEN GOTO 30
0
 319 FAST
 320 FOR N=1 TO 20
 325 IF A$(A,N)="
                 " THEN LET A$(A
,N)="............................
....."
 330 PRINT A$(A,N)
 335 NEXT N
 340 FOR N=0 TO 20
 342 IF N=20 THEN GOTO 352
 343 IF N<9 THEN PRINT AT N,31;N
+1
 345 IF N>=9 THEN PRINT AT N,30;
N+1;
 350 NEXT N
 352 SLOW
 355 PRINT AT 20,0;"ZEILENNUMMER
 DER KORREKTUR?"
 357 GOSUB 700
 360 LET N=VAL C$
 362 PRINT AT 20,0;"NEUER TEXT?
                  "
 364 GOSUB 700
 368 LET A$(A,N)=C$
 370 PRINT AT N-1,0;A$(A,N)
 400 GOTO 355
 430 PRINT "WENN SIE AUFNAHMEBER
EIT SIND"
 435 PRINT "DRUECKEN SIE BITTE >
P<"
 440 GOSUB 700
 445 IF C$<>"P" THEN GOTO 435
 450 CLS
 455 SAVE "█"
 460 GOTO 10
 500 LET N=1
 510 PRINT "NUMMER DER SEITE?"
 520 GOSUB 700
 525 LET A=VAL C$
 530 CLS
 535 IF A>15 OR A<1 THEN GOTO 51
0
 540 PRINT AT 0,0;"SEITE:";A;" Z
EILE:";N
 545 PRINT AT N,0;F$
 550 PRINT AT 20,0;"TEXT?
>>> NEUE SEITE>Y<"
 560 INPUT C$
 562 IF C$="X" OR C$="Y" THEN CL
S
 565 IF C$="X" THEN GOTO 10
 568 IF C$="Y" THEN GOTO 500
 570 LET A$(A,N)=C$
 590 PRINT AT N,0;A$(A,N)
 600 LET N=N+1
 610 IF N>20 THEN PRINT AT 20,0;
"NEUE SEITE ANFANGEN"
 620 IF N>20 THEN GOTO 500
 630 GOTO 540
 700 INPUT C$
 710 IF C$="X" THEN CLS
 720 IF C$="X" THEN GOTO 10
 730 RETURN
```

① **Listing »Notizblock«**: Damit lassen sich auf 15 Bildschirmseiten Texte mit je 20 Zeilen à 32 Zeichen unterbringen; beim Überschreiten der Zeilenlänge muß die nächste Zeile aufgerufen werden

Gibt man vom Menü (Bild 2) aus »17« ein, schaltet der Computer in den Korrekturmodus. Hierbei werden am rechten Bildrand die Zeilenzahlen dargestellt. Dann fragt der Computer nach der zu korrigierenden Zeile, und der neue Text kann geschrieben werden. Zurück ins Menü gelangt man auch hier durch Eingabe von »X«.
Mit »18« schließlich werden das Programm und die Datei auf Kassette gespeichert. Danach steht das Programm für neue Daten zur Verfügung.

Lothar Stöbener



② **Inhaltsverzeichnis:** Die Kopfzeilen der 15 Seiten (hier noch leer) wahren den Überblick. Das Menü (unten) dient als Gedächtnisstütze

Ein ROM für Tüftler

»Machen Sie aus Ihrem ZX 81 einen neuen Computer« heißt es in der Werbung für das »Aszmic«-ROM. Wir haben nachgesehen, zu welchen Taten der ZX 81 mit diesem Assembler-Betriebssystem fähig ist.

Der einfachste Weg, das Aszmic-ROM in Betrieb zu nehmen, ist der: Raus mit dem Sinclair-ROM und rein mit dem neuen ROM. Zwar ist es auch möglich, beide ROMs zugleich zu adaptieren, und das gerade benötigte mit einem Umschalter zu aktivieren, doch ist die Anleitung dafür im mitgelieferten Handbuch so gehalten, daß man schon eine gehörige Portion Vorkenntnisse zur Verwirklichung des Vorhabens benötigt.

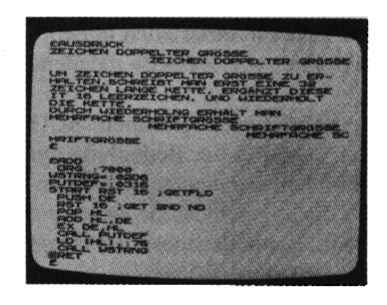
Basic wird zur Fremdsprache

Mit dem Einsetzen des neuen ROMs verliert der ZX 81 vollständig seine gewohnten Eigenschaften. Er beherrscht dann Assemblersprache, versteht keinen einzigen Basic-Befehl mehr und vergißt sämtliche Grafikzeichen. Der Bildschirm hat nunmehr 32 Zeilen mit 36 Spalten und der Cursor blinkt ständig.
Die Vorteile des ROMs treten nahezu ausschließlich für den erfahrenen Computeranwender zu Tage, der sich von seinem ZX 81 mehr erwartet als »nur« Basic: Er kann mit der im Vergleich zu Hexcode-Eingaben komfortablen Assembler-Programmierung leicht Maschinenprogramme entwickeln und hat dank einer wirkungsvollen Editierung die Möglichkeit, schnell und einfach Maschinenprogramme zusammenzustellen und zu ändern. Es lassen sich allerdings auch mit einigen Tricks, die im Handbuch erläutert sind, von Kassette geladene Basic-Programme damit bearbeiten.
Es gibt mit dem Aszmic-ROM zwei Betriebsarten des Rechners: EDIT und DEBUG. Diese unterscheiden sich schon in der Bedeutung der Taste NEWLINE: Im EDIT-Modus wird damit nur eine neue Zeile angefangen, während im DEBUG-Modus die abgeschlossene Zeile dem Rechner zur Ausführung übergeben wird. Beide Betriebsarten bieten

die REPEAT-Funktion, d. h. bleibt man länger als eine halbe Sekunde auf einer Taste, so werden acht Zeichen pro Sekunde geschrieben bzw. die Cursor-Funktionen achtmal pro Sekunde ausgeführt.
Wie der Name EDIT schon sagt, ist es in dieser Betriebsart ausschließlich möglich, zu editieren *(Bild 1)*. Editieren bedeutet in diesem Falle das Eintippen, Ändern oder Löschen von Maschinen- bzw. Basicprogrammen oder beliebigen anderen Texten. Die Daten werden dabei in »Files« (Dateien) eingeschrieben. Ein File ist ein benannter Speicherbereich zwischen einer Anfangs- und einer Endmarke, in dem sich beliebige Daten befinden können. Es ist also völlig gleichgültig, ob sich darin Assembler- oder Basic-Programme oder z. B. nur irgendwelche Namenslisten befinden. Die Files ermöglichen es, mehrere Programme im Speicher zu haben, und diese einzeln mit Hilfe der Filenamen ansprechen, abspeichern oder laden zu können.
Sämtliche Editierfunktionen sind über die Taste SHIFT und eine weitere Taste aufzurufen.

Der EDIT-Modus führt schnell zur gewünschten Textstelle

○ Cursor Funktionen: Der Cursor läßt sich in alle vier Richtungen verschieben, wobei hier die REPEAT-Wirkung ein besonderer Vorteil ist. Ein anderer Befehl setzt den Cursor in die oberste Zeile des Anzeigebereichs (Topline), die eine besondere Bedeutung hat.
Damit man ein längeres Programm schnell »überfliegen« kann, gibt es zwei Befehle, die den angezeigten Bereich eines Textes um 27 Zeilen nach oben oder unten verschieben. Der Cursor springt dabei in die mittlere Zeile am Bildschirm.
○ Löschfunktionen: Zu dem gewöhnlichen ROBOUT erhält man einen weiteren elektronischen Radiergummi, der das Zeichen entfernt,



① **Bildaufbau im EDIT-Modus:** Programme oder Texte lassen sich mit 32 Zeilen à 36 Zeichen darstellen. Ein Befehl holt die oben oder unten angrenzende Bildseite auf den Schirm

auf dem der Cursor gerade steht. Da das dahinterliegende Zeichen nachrückt, löscht dieser RUBOUT-Befehl alle hinter der Cursorposition befindliche Buchstaben einer Zeile, sofern man länger auf der Taste bleibt. Schließlich ist es möglich, mit jeweils einem einzelnen Befehl eine ganze Zeile oder gar ein ganzes Textstück zu löschen.

○ Einkopier-Funktionen: Diese sogenannten »Merge«-Funktionen bieten das schnelle Einkopieren dreier verschiedener Textstücke an beliebigen Stellen eines vorliegenden Programms oder Textes. Auf diese Weise lassen sich z. B. vollständige Befehle oder Befehlsteile wie »LD A, . . .«, die immer wieder vorkommen, schnell mit einem Tastendruck schreiben.

○ Suchfunktion: Besonders praktisch ist es, ein Programm (oder einen Text) nach dem Auftreten einer bestimmten Zeichenfolge (z. B. fehlerhafter Name) von unten nach oben durchsuchen zu können. Dabei springt der Cursor jeweils auf die gesuchte Zeichenfolge im Programm. Das, wonach gesucht wird, ist in der Topline einzugeben.

Entwicklungshilfe für Maschinenprogramme

Die Betriebsart DEBUG (Fehlerbeseitigung) erweckt die Möglichkeiten des Aszmic-ROMs zum Entwickeln von Maschinenprogrammen. Zahlreiche Anweisungen erlauben sowohl das Übersetzen und Starten von Assemblerprogrammen als auch eine rasche Fehlersuche.

○ Assemblierbefehl: Er erlaubt das Assemblieren, d. h. das Übersetzen eines in einem bestimmbaren File stehenden Assemblerprogrammes in die Z-80-Maschinensprache. Eine in dem Befehl anzugebende Bedingungszahl entscheidet z. B., ob und wo ein Assemblerlisting erstellt werden soll, ob der Maschinencode erzeugt, oder ob ein zweiter Übersetzungsdurchgang gemacht werden soll.

Leider ist die Anwendung dieser Bedingungszahl nur knapp im englischsprachigen Handbuch erklärt. Hier fehlen Beispiele zur Anwendung des Assemblierungsbefehls auf externe Unter- und Bibliotheksprogramme. Es bleibt zu hoffen, daß dies in der angekündigten deutschen Übersetzung besser behandelt wird. Vorschrift ist, daß ein zu übersetzendes Assembler-Programm zu den Zilog-Richtlinien für den Z-80-Prozessor entspricht.

○ Auflisten und Ändern von Speicherzellinhalten: Mit einer Anweisung lassen sich Speicherzellinhalte ab einer bestimmbaren Adresse in hexadezimaler Schreibweise auf den Bildschirm holen *(Bild 2)*.

Kleine Änderungen im Maschinencode lassen sich durch eine andere Anweisung durchführen, die das Ändern einzelner Speicherzelleninhalte erlaubt.

Auch ist es möglich, die Inhalte ganzer Speicherbereiche in andere Speicherbereiche zu kopieren oder einen beliebig großen Speicherbereich mit einem beliebigen hexadezimalen Wert zu füllen (Reservieren von Speicherplatz).

○ Laufanweisungen zur Fehlersuche: Eine Anweisung erlaubt das normale Starten eines Maschinenprogramms, um festzustellen, ob ein Fehler auftritt. Für den Fall, daß einer vorhanden ist, läßt sich das Programm in verschiedene Blöcke teilen, an deren Ende die Ausführung unterbrochen und ein in der Topline stehender DEBUG-Befehl ausgeführt wird. Das kann z. B. ein Befehl sein, der einige Registerinhalte zur Anzeige auf dem Bildschirm bringt.

Hat man damit den Fehler in einem bestimmten Block eingekreist, so ermöglicht ein anderer Befehl das Lokalisieren der fehlerhaften Zeile durch schrittweises Abarbeiten des Programmes.

○ Direktanweisung: Ein Befehl läßt die direkte Ausführung einer Assembleranweisung zu, ohne diese erst in ein Programm zu packen und übersetzen zu müssen. Das ist das Gleiche, als würde man mit dem ursprünglichen ROM eine einzelne Basic-Anweisung wie PRINT 2*3 durch NEWLINE zur Ausführung bringen (Direkteingabe). Damit läßt sich eine fehlerhafte Anweisung endgültig erkennen oder ganz einfach ein Programmschritt auf seine Wirkung hin ausprobieren.

○ Kassettenoperationen: Da man mehrere Programme gleichzeitig (in verschiedenen Files) im Speicher haben kann, ist die SAVE-Anweisung etwas länger geworden und erhält neben dem Namen, unter



② **Bildaufbau im DEBUG-Modus:** Hier z. B. ein Assemblerlisting bestehend aus Adressen, Hex-Codes und den dazugehörigen Mnemonics

dem das Programm abgespeichert werden soll, noch den Namen des Files, in dem es steht.
Sinnvollerweise wurde zwischen dem Abspeichern des Programmnamens und dem Abspeichern der restlichen Daten eine 5 s dauernde Pause vorgesehen, die es z. B. zuläßt, ein abgespeichertes Basic-Programm mit einem Aszmic-Titel zu laden und anschließend mit dem komfortablen Editor zu bearbeiten. Auf ähnliche Weise, aber im Handbuch nicht gut beschrieben, kann man Maschinensprachelemente in ein Basic-Programm einbauen. Eine weitere Anweisung erlaubt das Laden eines Files von Kassette.
Geschickte Programmierer können mit systemeigenen Unterroutinen zum Abspeichern und Laden eines einzelnen Bytes auf/von Kassette die Übertragungsrate bei den Kassettenoperationen auf 625 Byte pro

```
700F CD4501   ONEP CALL KEYBRD ;READKEYBOARD (RAW) LD HL,(FRA
7012 23        INC HL
7013 227040    LD (FRAMES),HL ;BUMP FRAME COUNT
7016 061E      LD B,IDLE
7018 10FE      DJNZ $ ;YOU JUST KEEP ME HANGING ON
701A 1601      LD D,PIXSIZE
701C 06FF      LD B,RASTERS
701E 0E01      LD C,TOPS
7020 CDED01    CALL OFRM1 ;WRITEOUTA PART FRAME
7023 3E01      LD A,1
7025 320040    LD (MFLAG),A ;LET NMI INTERRUPT HANDLER KNOW I
7028 0E5D      LD C,NNN+NNN+1 ;SETUP C FOR G COMMAND
702A D9        EXX
702B C32006    JP RESTOR ;GO INTO MIDDLE OF G HANDLING
702E          ;
702E          ;SETUP DISPLAY FILE....
702E          ;
702E 210041   CLEAR LD HL,DSTART
7031 110241    LD DE,DSTART+2
7034 010024    LD BC,'2400
7037 3676      LD (HL), 76 ;/NL/
7039 222340    LD (DFILE),HL
703C 23        INC HL
703D 3600      LD (HL),0
703F EDB0      LDIR
7041 C9        RET
```

③ **Originalausdruck:** Das Aszmic-ROM erlaubt die Ausgabe von 64 Zeichen je Zeile. Beim ZX-Drucker leidet dann jedoch die Lesbarkeit des Ausdrucks erheblich

Sekunde erhöhen. Der Mic-Ausgang läßt sich (im Gegensatz zu Katalogangaben) jedoch nicht als RS-232-C-Ausgang nutzen.

○ Druckbefehl: Mit einem Befehl läßt sich der Text eines zu bestimmenden Files vom Sinclair-Drucker ausdrucken. Hierbei läßt sich durch Verändern eines bestimmten Bytes der Drucker von 32 auf 64 Zeichen pro Zeile umstellen. Die ohnehin schon schlechte Lesbarkeit des Druckbildes leidet darunter freilich nochmals *(Bild 3)*. Außerdem läßt sich eine beliebig große Schrifthöhe erzielen, wenn man folgenden Trick anwendet: Man schreibt (in ein File) eine 32 Zeichen lange Kette, ergänzt diese mit 16 Leerzeichen und wiederholt die 32-Zeichen-Kette. So erhält man doppelte Schriftgröße. Durch Wiederholen des Vorgangs erzielt man eine mehrfache Schriftgröße.

○ Fertige Unterroutinen: Nicht alle Einzelheiten muß der Assembler-Programmierer beim Verwenden des Aszmic-ROMs selbst lösen. Nützliche Eingabe/Ausgabehilfen sind ihm in Form von ca. 30 fertigen Unterroutinen vorgegeben. Diese Unterroutinen können sinnvoll nur durch ein Assemblerprogramm aufgerufen werden, das die nicht gerade einfachen »Übergabebedingungen« erstellt. So sind z. B. erst gewisse Register zu laden, bevor ein String durch eine solche Unterroutine auf dem Bildschirm angezeigt werden kann.

○ Grafikmöglichkeiten: Grundsätzlich ist die Anwendung hochauflösender Grafik gegeben. Im Anhang des Handbuches sind verschiedene Software-Unterprogramme angegeben, um Grafikelemente zu erhalten. Abgedruckt sind Programme zum Plotten eines Punktes, zum Ziehen einer Linie zwischen zwei Punkten und zum Entfernen dieser Elemente. Die Anwendung ist jedoch nicht einfach, sondern erfordert komplexe Steuerprogramme, die wie bei den Unterroutinen Übergabebedingungen herstellen müssen.

Nur Profis können die »Features« voll nutzen

Das von Profisoft, Osnabrück, für 168 DM angebotene ROM ist ein Hilfsmittel für den geübten ZX-81-Anwender, der sich bereits mit Maschinensprache befaßt hat. Dies ist auch daran zu erkennen, daß sich der Autor des Handbuches immer wieder auf das »sicher schon vorhandene Wissen« aus der Lektüre des Buches »Programmierung des Z 80« von Ronald Zaks beruft. An diesen Stellen leidet die Verständlichkeit des Handbuches für Einsteiger sehr. Sie sollten deshalb sorgfältig abwägen, ob sich die Anschaffung des ROMs nur wegen des

guten Editors oder der Verfügbarkeit hochauflösender Grafik lohnt. Dagegen wird der »Kenner« von Maschinensprache seine Freude damit haben.

Jürgen Zimmermann

Softwaretip:

Kleiner Ersatz für Disassembler

Wer gerne einen Blick hinter die Kulissen von Maschinencode-Programmen wirft, ist im allgemeinen auf die Unterstützung eines Disassemblers angewiesen, der dem Inhalt der einzelnen Speicherstellen die entsprechenden Mnemonics des Z-80-Befehlssatzes zuordnet. Dazu muß meist ein eigenes Programm in den geschützten Speicherbereich oberhalb von RAM-TOP geladen werden, um bei Bedarf als Maschinencode-Programm aufgerufen zu werden.
Oft ist ein Disassembler jedoch nicht unbedingt erforderlich, und es genügt bereits, die Adressen und die einzelnen Speicherinhalte in hexadezimaler Form aufgelistet zu erhalten.
Das nachfolgend beschriebene Programm *(Bild 1)* liefert einen derartigen Ausdruck über einen beliebig langen Speicherbereich. Es muß lediglich die Anfangs- und Endadresse eingegeben werden. Als Beispiel für einen solchen Speicherausdruck zeigt *Bild 2* den Inhalt der Programmzeile 10 (teilweise).
Im Eingabeteil des Programms (Zeile 20 bis 70) werden der Anfangs- und Endadresse die Variablen A1 und A2 zugewiesen und am ersten Bildschirmausdruck angezeigt. Der Ausleseteil (Zeile 80 bis 180) holt die auszulesenden Speicherinhalte, sorgt für die Bereitstellung der Variablen C und den Aufruf des Code-Umsetzers. Der Dez/Hex-Converter (Zeile 200 bis 300) schließlich übersetzt die Variable C und sorgt für deren Ausdruck in hexadezimaler Form. Dieser Converter kann auch unabhängig vom übrigen Programm verwendet werden und ermöglicht dann das Umsetzen auch längerer Dezimalzahlen, sofern sie der Variablen C zugewiesen werden.

Eine SCROLL-Funktion ist mit Hilfe der Zeile
85 IF PEEK 16442 = 2 THEN SCROLL möglich.

Karl Heinz Krehan

```
  10 PRINT TAB (3);"SPEICHERAUSD
RUCK (HEXDUMP)",,,
  20 PRINT "VON SPEICHERADRESSE:
";
  30 INPUT A1
  40 PRINT A1
  50 PRINT "BIS SPEICHERADRESSE:
";
  60 INPUT A2
  70 PRINT A2,,,
  80 FOR N=A1 TO A2
  90 PRINT N;
 100 LET C=N
 110 PRINT TAB (5);":..";
 120 GOSUB 200
 130 LET C=PEEK N
 140 PRINT TAB (12);":..";
 150 IF C<16 THEN PRINT "0";
 160 GOSUB 200
 170 PRINT "..:";CHR$ PEEK N
 180 NEXT N
 190 STOP
 200 LET D=1
 210 LET E=INT (C/D)
 220 IF E>15 THEN LET D=D*16
 230 IF E>15 THEN GOTO 210
 240 IF C<16 AND D=1 THEN GOTO 2
90
 250 PRINT CHR$ (28+E);
 260 LET C=C-E*D
 270 LET D=D/16
 280 GOTO 210
 290 PRINT CHR$ (28+C);
 300 RETURN
```

① **Listing:** Neben der Preisgabe von Speicherzelleninhalten bietet dieses Programm auch einen Dezimal/Hexadezimal-Konverter, den man noch anderweitig nutzen kann

```
      SPEICHERAUSDRUCK (HEXDUMP)

VON SPEICHERADRESSE:  16513
BIS SPEICHERADRESSE:  16528

16513:..4081:..F5..: PRINT
16514:..4082:..C2..:TAB
16515:..4083:..10..:(
16516:..4084:..1F..:3
16517:..4085:..7E..:?
16518:..4086:..82..:▙
16519:..4087:..40..:RND
16520:..4088:..00..:
16521:..4089:..00..:
16522:..408A:..00..:
16523:..408B:..11..:)
16524:..408C:..19..:;
16525:..408D:..0B..:"
16526:..408E:..38..:S
16527:..408F:..35..:P
16528:..4090:..2A..:E
```

② **Ausdruck:** Adressen (dezimal und hexadezimal), Speicherzelleninhalte und entsprechende ZX-81-Zeichen für den Anfang der Programmzeile 10

Software:

Potenzieren und Logarithmieren

Der ZX 81 kennt keine Befehle, um Potenzen mit negativer Basis zu berechnen. Auch Berechnungen mit dem Zehner-Logarithmus (log) lassen sich nicht ohne weiteres durchführen, da der Computer nur den natürlichen Logarithmus (ln) kennt. Das hier vorgestellte Programm löst diese Aufgaben und vermag zusätzlich noch Logarithmen mit beliebiger Basis zu berechnen.
Bis zur Zeile 70 wird die Auswahl des richtigen Programmteils vorgenommen, die von der Eingabe eines Kennbuchstabens abhängt:

A – Potenzieren
B – Logarithmieren (log)
C – Logarithmus X zur Basis Y

Die Potenzrechnung beginnt ab Zeile 100. Falls die Basis negativ ist, darf der Exponent kein Bruch sein. Außerdem darf er weder negativ

```
  20 IF INKEY$="" OR INKEY$<>"A"
 AND INKEY$<>"B" AND INKEY$<>"C"
 THEN GOTO 20
  30 IF INKEY$="A" THEN LET A=10
0
  40 IF INKEY$="B" THEN LET A=30
0
  50 IF INKEY$="C" THEN LET A=40
0
  60 CLS
  70 GOTO A
 100 PRINT "X HOCH Y:"
 110 PRINT ,,,,"X=";
 120 INPUT X
 130 PRINT X
 140 PRINT "Y=";
 150 INPUT Y
 160 PRINT Y
 170 IF X<0 AND Y<>INT Y OR X=0
AND Y<=0 THEN GOTO 500
 200 IF INT (Y/2)=Y/2 THEN GOTO
240
 220 LET Q=SGN X*(ABS X)**Y
 230 GOTO 250
 240 LET Q=(ABS X)**Y
 250 PRINT ,,,,X;" HOCH ";Y;"=";
Q
 260 GOTO 600
 310 PRINT ,,,,"X=";
 320 INPUT X
 330 PRINT X
 340 LET Y=10
 350 GOTO 465
 410 PRINT ,,,,"X=";
 420 INPUT X
 430 PRINT X
 440 PRINT "Y=";
 450 INPUT Y
 460 PRINT Y
 465 IF X<=0 OR Y<=1 THEN GOTO 5
00
 470 LET P=LN X*(1/LN Y)
 480 PRINT ,,,,"LOGARITHMUS ";"X
";" ZUR BASIS ";Y;"=";P
 490 GOTO 600
 500 PRINT ,,,,"FEHLER"
 600 INPUT A$
 610 IF A$="" THEN GOTO 5
```

Listing: Gesichert gegen Fehleingaben ermöglicht dieses Programm Potenzieren und Logarithmieren mit dem ZX 81

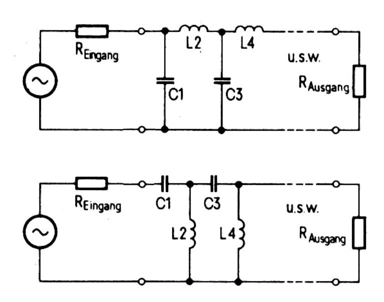
noch 0 sein, wenn die Basis den Wert 0 hat. Sobald eine dieser Vorschriften nicht beachtet wird, sorgt Zeile 170 dafür, daß »Fehler« angezeigt wird und das Programm zum Anfang zurückkehrt. Ist die Basis negativ, fällt das Ergebnis bei ungeradem Exponenten negativ aus, bei geradem Exponenten ist das Ergebnis positiv (Zeile 200).
Beim Logarithmieren wird in Zeile 470 davon Gebrauch gemacht, daß sich vom natürlichen Logarithmus jeder beliebige andere Logarithmus ableiten läßt. Zeile 465 verhindert dabei unzulässige Eingaben.

Sven Blicke

Software:

Berechnung von Butterworth-Hoch- und Tiefpässen

Die Butterworth-Funktion hat im Durchlaßbereich den flachsten Amplitudenverlauf. Dieser Verlauf wird gegenüber der Tschebyscheff-Funktion, die je nach Abfall im Übergangsbereich eine entsprechende Welligkeit aufweist, durch einen höheren Bauelementeaufwand erkauft.
Das vorliegende Programm ist eine Überarbeitung der Veröffentlichung »Hoch- und Tiefpaß-Berechnung in Basic« von E. Nolte in FUNKSCHAU 23/80. Nach Eingabe der geforderten Daten, das sind der Eingangs-(Ausgangs-)Widerstand, die Eckfrequenz (bei 3 dB Dämpfung definiert) und die Sperrfrequenz mit der ebenfalls einzugebenden Dämpfung in dB, erfolgt die Filterbezeichnung, und es werden Anzahl, Anordnung und die Werte der Filterelemente ausgegeben.
Bild 1 zeigt die jeweils benötigten Schaltungen, *Bild 2* das zugehörige Berechnungsprogramm. In *Bild 3* sind als Berechnungsbeispiel die



① **Prinzipschaltung** eines passiven Tief- (oben) und Hochpasses

Ergebnisliste eines Tiefpaßfilters mit den Daten: Grenzfrequenz (−3 dB) 35 MHz und Sperrdämpfung 80 dB bei der Sperrfrequenz 100 MHz wiedergegeben. Max Zaus

② **Listing:** Filterberechnung (Butterworth) mit dem ZX 81

```
  10 PRINT "[redacted]"
  20 PRINT
  30 PRINT "EINGABE EING./AUSG.-
WIDERSTAND R (IN OHM):";
  40 INPUT R
  50 PRINT R
  60 PRINT "EINGABE ECKFREQUENZ
(-3 DB) (IN HERTZ):";
  70 INPUT FC
  80 PRINT FC
  90 PRINT "EINGABE SPERRFREQUEN
Z FX     (IN HERTZ):";
 100 INPUT FX
 110 PRINT FX
 120 PRINT "EINGABE DAEMPFUNG A
(IN DB) BEI FX:";
 130 INPUT A
 140 PRINT A
 145 PRINT
 150 IF FC/FX>1 THEN GOTO 210
 160 LET N=A/(20*.4343*LN (FX/FC
))
 170 LET N=INT (N+1)
 180 LET E$="TP"
 190 PRINT "TIEFPASS-FILTER MIT
DER ELEMENT-ANORDNUNG : QUER-KON
DENSATOR,   LAENGS-SPULE.....UND
";N;" ELEMENTE"
 200 GOTO 250
 210 LET N=A/(20*.4343*LN (FC/FX
))
 220 LET N=INT (N+1)
 230 LET E$="HP"
 240 PRINT "HOCHPASS-FILTER MIT
DER ELEMENT-ANORDNUNG . LAENGS-K
ONDENSATOR, QUER-SPULE .....UND
";N;" ELEMENTE"
 250 DIM B(N)
 260 PRINT
 270 FOR I=N TO 1 STEP -1
 280 LET B(I)=(2*I-1)*PI/(2*N)
 290 NEXT I
 300 LET B=1/(PI*R*FC)
 310 LET C=-2
 320 FOR I=1 TO N
 330 LET I$=STR$ (I)
 340 PRINT I$;" ";
 350 GOSUB 440
 360 PRINT TAB 3;C$;
 370 IF E$="TP" THEN GOTO 490
 380 IF E$="HP" THEN GOTO 530
 390 LET C=C*(-1)
 400 LET RR=R**C
 410 LET B=B*RR
 420 NEXT I
 430 STOP
 440 IF C=2 THEN GOTO 470
 450 LET C$="C="
 460 RETURN
 470 LET C$="L="
 480 RETURN
 490 PRINT INT (1E12*SIN (B(I))*
B+.5)/1E6;
 500 GOSUB 570
 510 PRINT TAB 15;F$
 520 GOTO 390
 530 PRINT INT (1E12*B/SIN (B(I)
)/4)+.5/1E6;
 540 GOSUB 570
 550 PRINT TAB 15;F$
 560 GOTO 390
 570 IF C=2 THEN GOTO 600
 580 LET F$="MIKRO-FARAD"
 590 RETURN
 600 LET F$="MIKRO-HENRY"
 610 RETURN
```

[redacted]

```
EINGABE EING./AUSG.-WIDERSTAND R
 (IN OHM):50
EINGABE ECKFREQUENZ (-3 DB) (IN
HERTZ):35000000
EINGABE SPERRFREQUENZ FX     (IN
HERTZ):100000000
EINGABE DAEMPFUNG A (IN DB) BEI
FX:80

TIEFPASS-FILTER MIT DER ELEMENT-
ANORDNUNG : QUER-KONDENSATOR,
LAENGS-SPULE.....UND 9 ELEMENTE

1  C=.000032     MIKRO-FARAD
2  L=0.227364    MIKRO-HENRY
3  C=.000139     MIKRO-FARAD
4  L=0.427305    MIKRO-HENRY
5  C=.000182     MIKRO-FARAD
6  L=0.427305    MIKRO-HENRY
7  C=.000139     MIKRO-FARAD
8  L=0.227364    MIKRO-HENRY
9  C=.000032     MIKRO-FARAD
```

③ **Ausdruck:** Hier wurde ein Tiefpaß mit neun Elementen berechnet. Die obere Bildhälfte zeigt, wie im Dialogbetrieb die Eingabewerte angefordert werden

ZX 81 à la carte (1):

Messer und Gabel

Teil 1: 6-KByte-RAM-Erweiterung

Zu Beginn einer Bauanleitungsserie für den ZX 81 stellen wir das Grundbesteck vor: RAM-Erweiterung und I/O-Port. Beide sind gemeinsam auf einer Basisplatine untergebracht.

Mehr über die Bauanleitungsserie »ZX 81 à la carte« verrät der Kasten auf Seite 49. Hier im ersten Teil geht es um die Beschreibung der ZX-81-Schnittstelle und um die RAM-Erweiterung. Im Teil 2 folgt das Ein-/Ausgabe-Interface und die Platine für beide Baugruppen. Mit dieser Basisplatine läßt sich bereits eine Menge anfangen.

Das Tor nach draußen: die ZX-81-Schnittstelle

Der ZX 81 hat an seiner Rückseite eine Kontaktleiste, die normalerweise für eine Speichererweiterung und den Drucker vorgesehen ist. Wie ein Blick auf die Anschlußbelegung der Kontaktleiste zeigt *(Bild 1)*, sind hier tatsächlich alle für Hardware-Erweiterungen wichtigen Signale herausgeführt.

○ Betriebsspannung 9V/5V/0V (Masse): Als Ausgang verwendet liefert der 9-V-Anschluß die unstabilisierte Gleichspannung des Rechnernetzteils. Falls die Belastung nicht zu groß wird, läßt sich diese Spannung auch für externe Hardware verwenden. Andererseits kann über diese Leitung auch eine Stromversorgung des Rechners erfolgen, falls für umfangreiche Erweiterungen ohnehin ein anderes Netzteil erforderlich wird.
Die stabilisierte 5-V-Versorgung ist für externe Schaltungen nur bedingt geeignet, da bei zu hoher Belastung der 5-V-Regler im Rechner sehr heiß wird, und seine Temperaturschutz-Schaltung ansprechen kann.

○ Adreßleitungen A0. . .A15: Es steht der komplette Adreßbus der Z-80-CPU (CPU: Central Processing Unit – Zentraleinheit) zur Verfügung. Beim Verwenden von Adressen für externe Schaltungen muß geprüft werden, ob nicht gleichzeitig auch die internen Speicher angesprochen werden (wegen unvollständiger Adreßdecodierung). Die

Leitungen sind H-aktiv (H: high), das bedeutet, daß eine logische 1 durch einen hohen Spannungspegel realisiert wird.

○ Datenleitungen D0...D7: Hier kann auf den 8-Bit-Datenbus zugegriffen werden. Die Leitungen sind ebenfalls H-aktiv.

○ ROM CS/RAM CS (chip-select – Chip-Anwahl). Während eines Zugriffs der Zentraleinheit auf die internen Speicher gehen die Signale ROM CS bzw. RAM CS auf logisch 0 (low: L-aktiv). Da diese Leitungen intern durch Widerstände von den Ausgängen des Sinclair-Logik-Chips entkoppelt sind, kann durch ein von außen zugeführtes H-Signal der jeweilige Speicher abgeschaltet werden, ohne daß dieses IC Schaden nimmt.

○ Systemtaktfrequenz Φ: Die Taktfrequenz des Z 80 A beträgt 3,23 MHz. Dieses durch ein Keramikfilter stabilisierte Signal kann für Zähler- und Zeitmeßanwendungen extern verwendet werden.

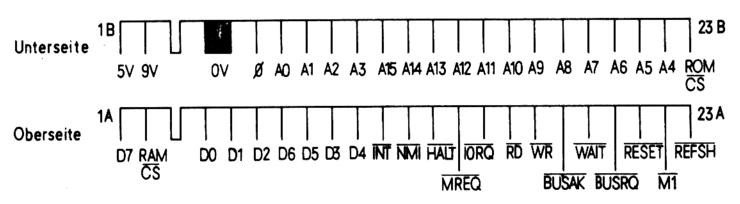
○ INT (interrupt request – Unterbrechungsaufforderung): Wenn INT auf logisch 0 geht, führt die CPU einen maskierbaren, d. h. durch einen Software-Befehl unterdrückbaren Interrupt aus. Da dieses Signal vom Betriebssystem verwendet wird, kann es von Erweiterungsschaltungen nicht sinnvoll benutzt werden.

○ NMI (nonmaskable interrupt):
Nichtmaskierbarer Interrupt (L-aktiv), ist ebenso wie INT nicht verwendbar.

○ HALT: Dieser L-aktive Ausgang zeigt an, daß die CPU einen Software-HALT-Befehl ausführt. Eine externe Verwendung ist beim ZX 81 ebenfalls nicht unproblematisch.

○ MREQ (memory request – Speicheraufforderung): Während eines Speicherzugriffs durch die CPU geht dieser Ausgang auf logisch 0.

○ IORQ (input/output-request – Ein-/Ausgabe-Aufforderung): Die Z-80-CPU hat spezielle Ein-/Ausgabe-Befehle, mit denen 256 verschiedene Ein-/Ausgabe-Adressen angesprochen werden. Dabei geht



① **ZX-81-Schnittstelle:** Die Signale INT, NMI, HALT, IORQ, BUSAK, BUSRQ, und M1 lassen sich für Hardwareerweiterungen nicht nutzen

$\overline{\text{IORQ}}$ auf logisch 0. Da diese Befehle auch für die Tastaturabfrage und Bildschirmausgabe des ZX 81 benutzt werden, ist eine externe Verwendung von $\overline{\text{IORQ}}$ nicht zweckmäßig.
- $\overline{\text{RD}}/\overline{\text{WR}}$ (read/write – lesen/schreiben): Diese Ausgänge (L-aktiv) zeigen, ob die CPU einen Lese- bzw. Schreibzugriff auf einen Speicher (bzw. I/O-Baustein) ausführt. Daher ist gleichzeitig auch $\overline{\text{MREQ}}$ (bzw. $\overline{\text{IORQ}}$) aktiv.
- $\overline{\text{BUSRQ}}/\overline{\text{BUSAK}}$ (bus request/bus acknowledgement – Busanforderung/Quittung): Durch ein L-Signal auf dem $\overline{\text{BUSRQ}}$-Eingang werden Adreß-, Daten- und Steuerbus von der CPU freigegeben (Bestätigung durch L-Signal auf dem $\overline{\text{BUSAK}}$-Ausgang) und können extern verwendet werden. Diese Leitungen sind beim ZX 81 nicht sinnvoll nutzbar.
- $\overline{\text{WAIT}}$ (warten): Diese Eingangsleitung (L-aktiv) kann benutzt werden, um langsame Speicher- oder I/O-Bausteine einsetzen zu können. Solange $\overline{\text{WAIT}}$ aktiv ist, wartet die CPU.
- $\overline{\text{RFSH}}$ (refresh – auffrischen): Dieses Signal (L-aktiv) ist für den Auffrisch-Zyklus bei dynamischen Speichern wichtig.
- $\overline{\text{M1}}$: Während eines Befehl-Lesezyklusses wird dieses Signal logisch 0. Für Erweiterungen ist es nicht sinnvoll anwendbar.
- $\overline{\text{RESET}}$ (rücksetzen): Durch ein L-Signal wird die CPU auf Adresse 0 rückgesetzt. Dadurch wird leider auch ein im ZX 81 vorhandenes Programm gelöscht. Aus- und erneutes Einschalten der Spannungsversorgung hat dieselbe Wirkung.

Werden die Ausgangssignale des ZX 81 verwendet, so muß beachtet werden, daß die Belastung nicht zu groß wird. Aus diesem Grund ist das Zwischenschalten von Verstärkern (Pufferung) sinnvoll.

Ein Adreßdecoder plaziert die Speicherblöcke

In der Grundversion hat der ZX 81 nur eine geringe Speicherkapazität von 1 KByte RAM. Mit zusätzlichen 6 KByte RAM ist man für viele Anwendungsfälle gerüstet. Durch Verwenden von Speicher-ICs des Typs HM 6116 (2K × 8 Bit) kann der Speicher auch in Einzelschritten von jeweils 2 KByte ausgebaut werden. Damit der Computer die Speichererweiterung auch nutzen kann, muß sich der externe RAM-Bereich nahtlos an den internen Bereich anschließen. Daher ist es nötig, sich die Speicherorganisation des ZX 81 für eine 6-KByte-RAM-Erweiterung klar zu machen *(Bild 2)*.

Eine Eigentümlichkeit dieses Computers ist die dezimale Angabe von Adressen und Speicherinhalten am Bildschirm. Im Bild sind die Adressen sowohl dezimal als auch hexadezimal aufgeführt. Die vierte Spalte zeigt die dazugehörigen logischen Pegel auf den Adreßleitungen.
Da die Z-80-CPU bei RESET den ersten (Maschinen-)Programmbefehl auf Adresse 0 erwartet, liegt das ROM für Betriebssystem und Basic-Interpreter (8 KByte) zwischen 0 und 8191. Überraschenderweise erscheint es nochmals zwischen 8192 und 16383. Der Grund dafür ist eine unvollständige Adreßdecodierung, d.h. für die Zuordnung der Speicheradressen im ROM zu den von der CPU ausgegebenen Adreßsignalen werden nur die Leitungen A0...A12 verwendet. Dadurch »sieht« das ROM z.B. 0 und 8192 als dieselbe Adresse. Man kann sich dies an den dezimalen Zahlen 1293 und 7293 klar machen: Betrachtet man nur die letzen drei Ziffern, so erhält man beide Male dieselbe Zahl, nämlich 293.

HEX	DEZ	Y	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
5FFF	24575	Y7	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	Adreß-bereich-für I/O
5C00	23552		0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	
5BFF	23551	Y6	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	6116 III
5800	22528		0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	
57FF	22527	Y5	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	
5400	21504		0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	
53FF	21503	Y4	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	6116 II
5000	20480		0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	
4FFF	20479	Y3	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	
4C00	19456		0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4BFF	19455	Y2	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	6116 I
4800	18432		0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	
47FF	18431	Y1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	
4400	17408		0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	
43FF	17407	Y0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1K internes RAM
4000	16384		0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3FFF	16383		0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	ROM Doppel
2000	8192		0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
1FFF	8191		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	ROM
0000	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

② **Speicherorganisation:** Die 6-KByte-RAM-Erweiterung nimmt den Adreßbereich 17408 bis 23551 in Anspruch. Y0 bis Y6 sind die Signale eines Adreßdecoders zum Auswählen von Speicherblöcken (siehe Text). Y7 gibt den I/O-Port (Teil 2) frei

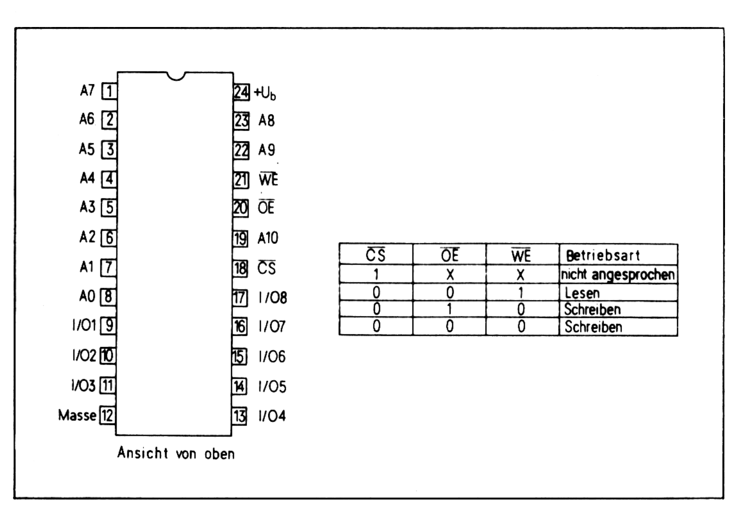
Von 16384 bis 17407 schließt sich das interne RAM an. Die RAM-Obergrenze wird beim Einschalten vom Computer mit einem Testprogramm ermittelt und als Systemvariable RAM-TOP gespeichert (deshalb muß auch eine Speichererweiterung nahtlos ans interne RAM anschließen). Durch die Befehlseingabe

PRINT PEEK 16388 + 256 * PEEK 16389

erhält man ihren Zahlenwert in dezimaler Schreibweise (siehe auch FS 11/83, Seite 78). Er stellt die Adresse des ersten nicht existierenden Bytes dar; ohne Speichererweiterung ergibt sich also die Zahl 17408.

In *Bild* 3 sind das Anschlußschema und die Funktionstabelle eines Speicherbausteins HM 6116 gezeigt. Dieser Speicher erfordert elf Adreßleitungen (A0. . .A10). Wenn man ihn direkt mit den entsprechenden Leitungen des Adreßbusses verbindet, so würde er gemeinsam mit dem internen ROM zwischen Adresse 0 und 2047 (2 KByte) angesprochen. Erwünscht ist er allerdings zwischen den Adressen 17408 und 19455.

Erreichen läßt sich das, indem man die Adreßleitungen A11. . .A14 zur Unterscheidung heranzieht. Im Bereich von 16384 bis 24575 ist stets A14 = 1 und A13 = 0. Betrachtet man weiterhin A12, A11 und A10 als dreistellige Binärzahl, so entspricht jeder solchen Zahl ein



$\overline{CS}$	$\overline{OE}$	$\overline{WE}$	Betriebsart
1	X	X	nicht angesprochen
0	0	1	Lesen
0	1	0	Schreiben
0	0	0	Schreiben

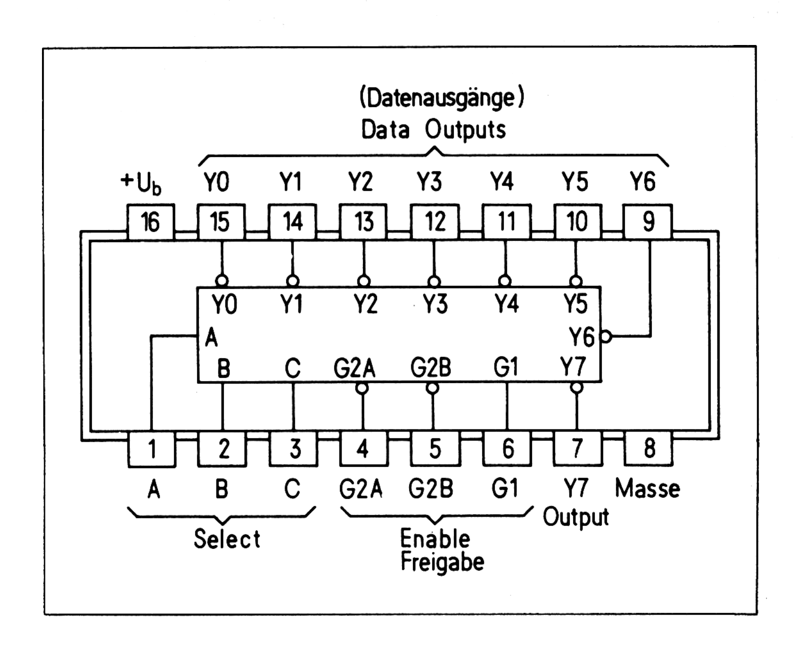
③ **Speicher-IC:** Der Speicherbaustein HM 6116 ist ein statisches 2 KByte RAM (2 K × 8). X bedeutet: Signalpegel gleichgültig

1 KByte großer RAM-Bereich. Ein 3-zu-8-Decoder 74LS138 kann in Abhängigkeit von dieser Zahl (C B A in *Bild 4*) jeweils einen seiner Ausgänge Y0. . .Y7 auf logisch 0 schalten (Spalte 3 in Bild 2).
Führt man seinem Freigabe-Eingang G1 (H-aktiv) das Signal A14, den Eingängen G2A und G2B (L-aktiv) die Signale $\overline{\text{MREQ}}$ und A13 zu, so ist gewährleistet, daß der Decoder nur während eines Speicherzugriffs auf die Adressen 16384 bis 24575 angesprochen wird.
A15 wird nicht benutzt. Das hat zur Folge, daß sich die gesamte Speicheraufteilung zwischen den Adressen 32768 und 65535 wiederholt (unvollständige Adreßdecodierung).

Da der 3-zu-8-Decoder jeweils 1-KByte-Blöcke auswählt, die Speicher-ICs aber zwei KByte umfassen, müssen jeweils zwei Ausgangsleitungen über eine ODER-Verknüpfung zusammengefaßt werden, um den jeweiligen Speicher anzusprechen. Das bedeutet, Y1 oder Y2 aktivieren den $\overline{\text{CS}}$-Eingang (siehe Bild 3) des ersten Speicher-ICs (logisch 0 an $\overline{\text{CS}}$ aktiviert das IC); Y3 oder Y4 bzw. Y5 oder Y6 bewirken dasselbe für die anderen Speicher.
Wenn Y0 = 0 ist, ist das interne RAM an der Reihe. Das bedeutet umgekehrt, daß während Y0 = 1 externe Bausteine angesprochen werden. Daher ist dieses Signal zum Abschalten des internen RAMs geeignet. Der durch Y7 = 0 erfaßte Bereich wird für die in Teil 2 beschriebene Schaltung benutzt.

Die Schaltung der Speichererweiterung

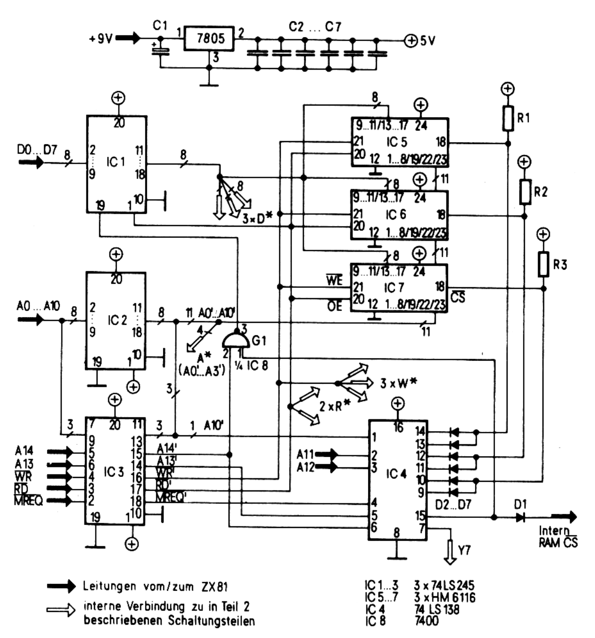
Bild 5 zeigt das Schaltbild der 6-KByte-Speichererweiterung. Die Stromversorgung erfolgt über den 9-V-Ausgang des ZX 81 und einen Spannungsregler 7805. Zum Abblocken von Impulsstörungen auf den Versorgungsleitungen dienen die Kondensatoren C1. . .C7.
Alle vom Computer kommenden Signale werden mit Bustreibern 74LS245 (IC1 bis IC3) gepuffert (Ausnahme A11 und A12, die beide nur eine LS-TTL-Last zu treiben haben).
A0. . .A10 gehen nach der Pufferung direkt an die Anschlüsse der Speicher-ICs (IC5 bis IC7). Die Leitungen A14', A13' und $\overline{\text{MREQ'}}$ aktivieren den Adreßdecoder 74LS138 (IC4) wie zuvor erläutert. In Abhängigkeit von dem aus A10', A11 und A12 gebildeten Eingangswort wird eine seiner acht Ausgangsleitungen auf logisch 0 geschaltet (siehe Bild 4).
Die ODER-Verknüpfung erfolgt für jeweils zwei dieser Ausgangsleitungen über die Germanium-Dioden D2. . .D7. Silizium-Dioden sind wegen ihrer höheren Flußspannung von 0,7 V ungeeignet, da dann eine logische 0 Spannungswerte bis zu 1,1 V erreichen kann (maximale Ausgangsspannung für logisch 0 bei TTL-Gattern 0,4 V zuzüglich 0,7 V), ein Speicher-IC am $\overline{\text{CS}}$-Eingang aber nur maximal 0,8 V als logisch 0 akzeptiert.
Das $\overline{\text{RD}}$'-Signal wird dem Anschluß $\overline{\text{OE}}$ (output enable – Freigabe der Ausgänge zum Lesen), das $\overline{\text{WR}}$'-Signal dem Anschluß $\overline{\text{WE}}$ (write enable – Freigabe zum Einschreiben von Daten) von IC5 bis IC7 zugeführt (siehe Bild 3).



Eingänge / Inputs					Ausgänge							
Freigabe		Select										
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	1	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	1	1	0

④ **Adreßdecoder:** Das IC 74LS138 ist ein 3-zu-8-Decoder. Abhängig von den Eingangssignalen C, B, A, nimmt einer der Y-Ausgänge 0-Pegel an. G1 und G2* sind Freigabe-Eingänge, wobei G2* die UND-Verknüpfung der Signale G2A und G2B ist

Um zu verhindern, daß das interne RAM gleichzeitig mit dem externen Speicher auf den Datenbus zugreift, wird es während Y0 = 1 über die Diode D1 abgeschaltet. Solange Y0 = 0 oder A14 = 0 ist (siehe Bild 2) wird der Datenbus vom internen RAM bzw. ROM belegt. Dann wird der Datenbustreiber IC1 abgeschaltet (der Tri-State-Ausgang wird hochohmig), um einen Doppelzugriff zu vermeiden. Die hierfür nötige ODER-Verknüpfung erfolgt über das NAND-Gatter G1, das in negativer Logik (Y0 und A14 sind hier L-aktiv) eine ODER-Funktion realisiert.
Da der Datenbuspuffer IC1 in Abhängigkeit von einem Schreib- bzw. Lesezugriff die Daten aus dem Computer heraus oder in den Computer hinein leiten soll, muß seine Durchlaßrichtung über die $\overline{\text{RD}}$’-Leitung umgeschaltet werden (Pin 1 von IC 1): $\overline{\text{RD}}$’ = 0 bedeutet Lese-Operation, $\overline{\text{RD}}$’ = 1 bedeutet Nichtlese-Operation. Die zu den Punkten D*, A*, W* und R* weisenden Pfeile beziehen sich auf den Schaltungsteil, der in Teil 2 besprochen wird. Oskar Merker



⑤ **6-KByte-RAM-Erweiterung:** Die Platine zu dieser Schaltung folgt in Teil 2

ZX-81-Softwaretip:

Abfragen des Speichers

Mit folgender Eingabe läßt sich herausfinden, wieviel Platz ein Programm im Speicher einnimmt:

PRINT PEEK 16396+256* PEEK 16397–16509

Erläuterung: Ein Programm beginnt immer ab Adresse 16509. Die Systemvariable D-FILE unter den Adressen 16396 und 16397 gibt dagegen die Adresse an, mit der der Bildschirmbereich anfängt, der sich direkt an den Programmbereich anschließt (siehe auch Handbuch, Kapitel 27). Daher ergibt die Differenz zwischen D-FILE und 16509 den Programmumfang.
Auf die gleiche Weise erhält man den Speicherumfang des Variablenbereichs als Differenz von E-LINE und VARS:

PRINT PEEK 16404-PEEK
16400+256* (PEEK 16405-PEEK 16401)-1

Diesmal müssen die Inhalte von zwei Systemvariablen berücksichtigt werden, weil sowohl Anfang als auch Ende dieses Speicherbereichs nicht festliegen. Die Eins muß abgezogen werden, weil grundsätzlich ein Byte mit Inhalt 128 im Variablenbereich vorhanden ist.
Den Umfang des noch freien Speicherbereichs, vielleicht die interessanteste Information beim Programmieren, erfährt man näherungsweise nach folgender Eingabe:

PRINT PEEK 16386-PEEK
16404+256*(PEEK 16387-PEEK 16405)

Hierbei wird die Differenz zwischen ERR-SP und E-LINE ermittelt. E-LINE sagt aus, wo der noch ungenutzte Speicherbereich beginnt. ERR-SP gibt an, wo der GOSUB-Stapel anfängt, wodurch das Ende des freien Speicherbereichs in etwa gegeben ist. Michael Schramm

Messer und Gabel

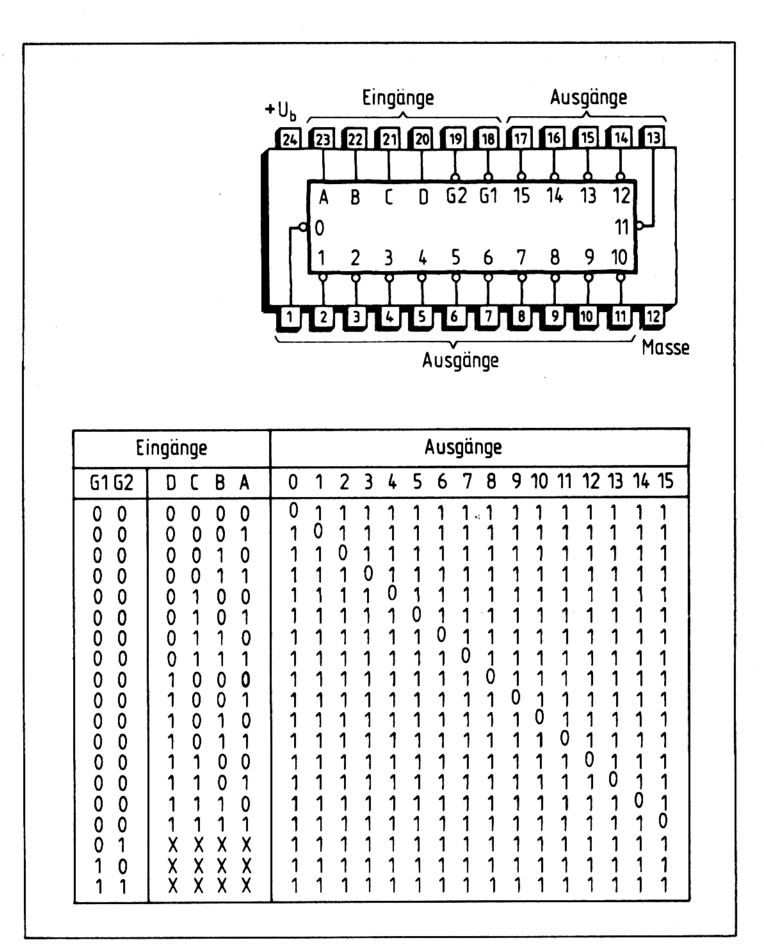
Teil 2: I/O-Port und Softwareschalter

Die im vorangegangenen Teil begonnene Beschreibung der Basisplatine wird jetzt abgeschlossen. Damit stehen die RAM-Erweiterung und das I/O-Interface zur Anwendung bereit.

Der Nachteil der ZX-81-Schnittstellen für die Tastatur, den Drucker, Bildschirm und den Kassettenrecorder ist, daß kein direkter Kontakt zum Umfeld besteht; stets muß der Mensch als Vermittler zusätzlich in Aktion treten. Wünschenswert wäre aber eine direkte Verbindung, um z.B. ein Gerät unmittelbar durch den Computer ein- oder ausschalten zu können. Umgekehrt kann auf diesem Weg eine unmittelbare Datenerfassung, z.B. der Zimmertemperatur, erfolgen.
Für diesen Zweck stehen eine Reihe spezieller ICs zur Verfügung, wie der Z-80-PIO (Parallel In/Out). Bei diesem IC ist jedoch ein Programmieren der verschiedenen Betriebsarten erforderlich. Dem Anfänger erschwert das die Bedienung des I/O-Ports. Daher wurde im folgenden ein anderes Konzept verwendet.

Mit PEEK und POKE werden Daten ein- und ausgegeben

Speicherstellen können in Basic mit dem Befehl PRINT PEEK (Adresse) gelesen, mit POKE (Adresse, Zahl) geladen werden. Ersetzt man das Speicher-IC durch ein Port-IC, kann auf die gleiche Weise eine Ein- oder Ausgabe realisiert werden. Daher schließt sich der I/O-Bereich an den Adreßbereich der Speichererweiterung an.
Zum Adressieren des Ports steht vom Adreßdecoder 74LS138 (IC 4 der Speichererweiterung) das Signal Y7 zur Verfügung, das im Adreßbereich 23 552. . .24 575 logisch 0 ist (siehe Teil 1). Mit diesem Signal läßt sich ein 4-zu-16-Decoder 74LS154 über die Eingänge G1/G2 aktivieren *(Bild 1)*. Werden an seine Eingänge A, B, C und D die Adreßleitungen A0', A1', A2' und A3' angelegt, so wählt er in Abhängigkeit von der Adreßkombination eine seiner 16 Ausgangsleitungen an



Eingänge						Ausgänge															
G1	G2	D	C	B	A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	1	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	X	X	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

① **4-zu-16-Decoder 74LS154:** Die vier Eingangssignale A bis D bestimmen, welcher der 16 Ausgänge logisch 0 annimmt

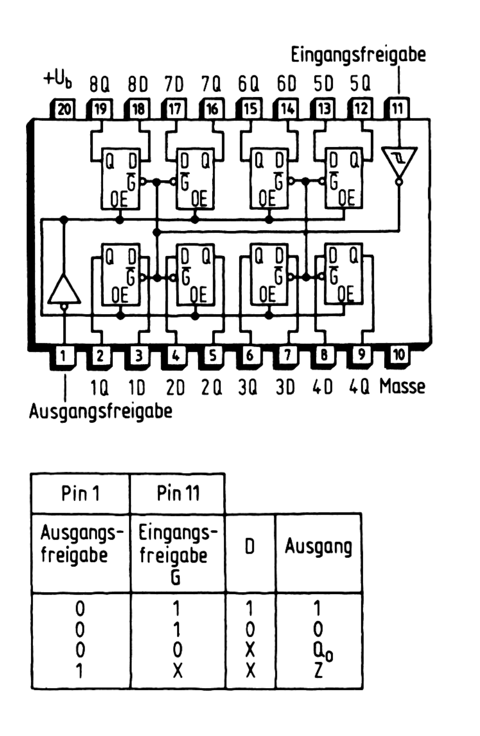
(L-aktiv, logisch 0). Auf diese Weise können die Adressen 23 552. . . .23 567 einzeln angesprochen werden (die Basisplatine nutzt vorerst nur die drei Adressen 23 556 bis 23 558).
Wegen unvollständiger Adreßdecodierung wiederholt sich das Ansprechen dieser Adressen 64mal bis zur Adresse 24 575, was aber nicht weiter stört.

Der Datenbus: Pipeline zum I/O-Port

Beim Ansprechen der Port-Adresse (23 558) müssen zur Dateneingabe die von außen kommenden Daten auf den Datenbus gelegt werden. Hierfür läßt sich sehr gut der Bus-Treiber 74 LS 245 zweckentfremden *(Bild 2)*.

Legt man seinen Anschluß 1 an Masse (logisch 0), so ist eine Datenübertragung von außen (B) zum Datenbus (A) möglich. Dies geschieht allerdings nur, wenn an Pin 19 (Enable-Freigabe) ein L-Signal (logisch 0) anliegt. Ist das Signal an Pin 19 logisch 1, so sind die IC-Ausgänge hochohmig (Tristate) und belasten den Datenbus nicht.

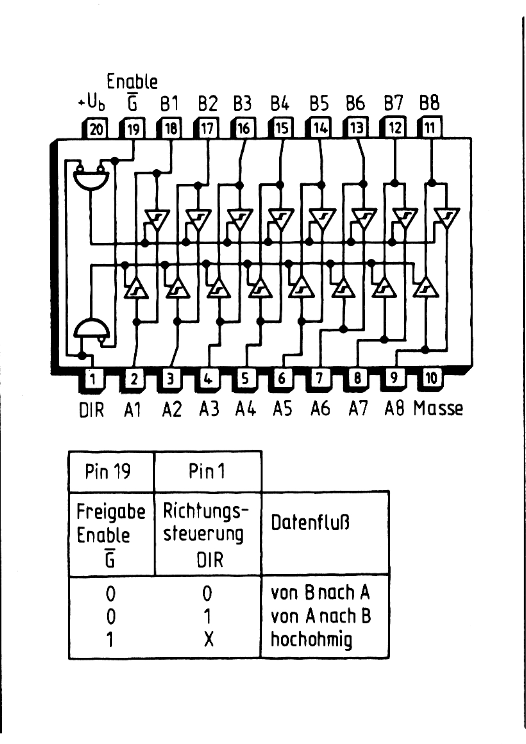
Die Eingabe-Daten müssen vorhanden sein, wenn die Port-Adresse angesprochen wird, denn ein Zwischenspeichern erfolgt nicht. Dies ist in den meisten Anwendungsfällen allerdings kein Nachteil.



Pin 1 Ausgangsfreigabe	Pin 11 Eingangsfreigabe G	D	Ausgang
0	1	1	1
0	1	0	0
0	0	X	Q_0
1	X	X	Z

② **Bustreiber 74LS245:** Er läßt sich gut zur Dateneingabe von der Außenwelt (B) zum Datenbus (A) zweckentfremden, wenn Pin 1 mit Masse verbunden wird

Da andererseits bei der Datenausgabe die Daten von der CPU nur sehr kurze Zeit auf den Datenbus gebracht werden (einige Mikrosekunden lang), ist in diesem Fall ein Zwischenspeichern erforderlich.
Diese Aufgabe kann ein 8fach-Latch 74LS373 (Latch – Zwischenspeicher) übernehmen *(Bild 3)*. Seine Ausgänge werden durch ein L-Signal an Pin 1 aktiviert, ein H-Signal (logisch 1) schaltet die Ausgänge in den hochohmigen Zustand. H-Pegel an Pin 11 (Enable) veranlaßt das IC, die an den Eingängen anliegenden Daten zu übernehmen und zu speichern. Dieses Signal muß also vom Computer geliefert werden, sobald die Ausgabe-Daten auf dem Datenbus vorhanden sind.



Pin 19	Pin 1	
Freigabe Enable G̅	Richtungssteuerung DIR	Datenfluß
0	0	von B nach A
0	1	von A nach B
1	X	hochohmig

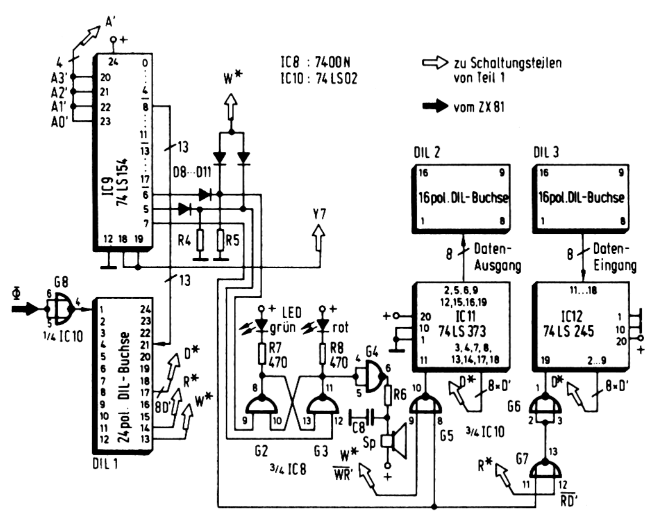
③ **8-Bit-Zwischenspeicher:** Das IC 74LS373 speichert die nur kurzfristig auf dem Datenbus liegende Information Q_0: alte Daten: Z: Ausgänge hochohmig

Schaltung des I/O-Ports

In *Bild 4* ist die Port-Schaltung gezeigt. Sie ergänzt die Schaltung der Speichererweiterung aus dem vorangegangenen Teil, daher sind die Bauteile fortlaufend durchnumeriert.

Im Adreßbereich 23552...24575 wird IC 9 (4-zu-16-Decoder) aktiviert, die Adressen A0'...A3' wählen dann wie erläutert eine der 16 Ausgangsleitungen an. Wenn dabei die Port-Adresse 23558 angesprochen wird, führt Pin 7 L-Signal. Falls gleichzeitig $\overline{\text{WR'}}$ logisch 0 ist (wenn z.B. ein POKE-Befehl gegeben wurde), wird über das NOR-Gatter G 5 (da negative Logik: UND-Verknüpfung) das 8fach-Latch IC 11 angesprochen (H-Pegel an Pin 11), und es speichert die auf dem Datenbus liegenden Informationen. Sie stehen dann an der Buchse DIL 2 extern zur Verfügung.

Falls $\overline{\text{RD'}}$ und Pin 7 von IC 9 ein L-Signal führen (z.B. bei einem PEEK-Aufruf) wird auf gleiche Weise über G 7 das IC 12 adressiert. G 6 dient als Inverter, da IC 12 an Pin 19 in diesem Fall L-Pegel benötigt. Dadurch werden die an DIL 3 von außen kommenden Daten in den Computer übernommen.



④ **I/O-Port und Softwareschalter:** Mit der 6-KByte-RAM-Erweiterung aus Teil 1 (die Pfeile führen zu dort angegebenen Schaltungspunkten) ist das Schaltbild der Basisplatine jetzt komplett

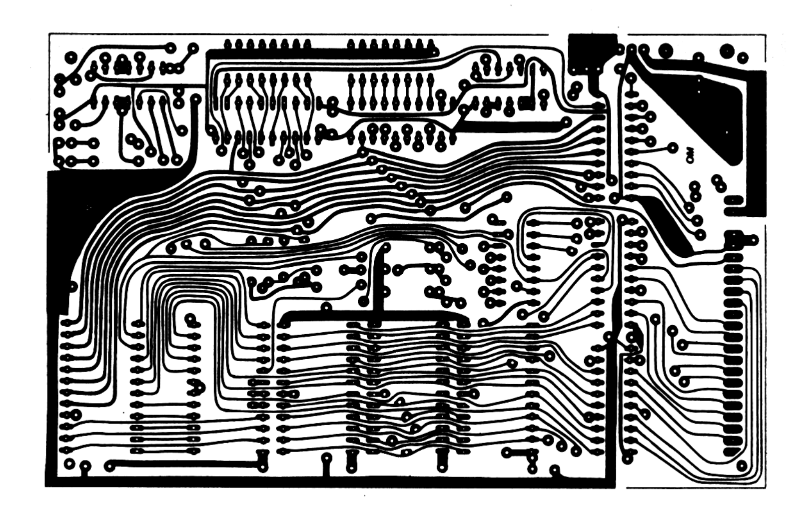
23 556 Einschalten LED grün
23 557 Einschalten LED rot
23 558 I/O-Port (jeweils unidirektional)

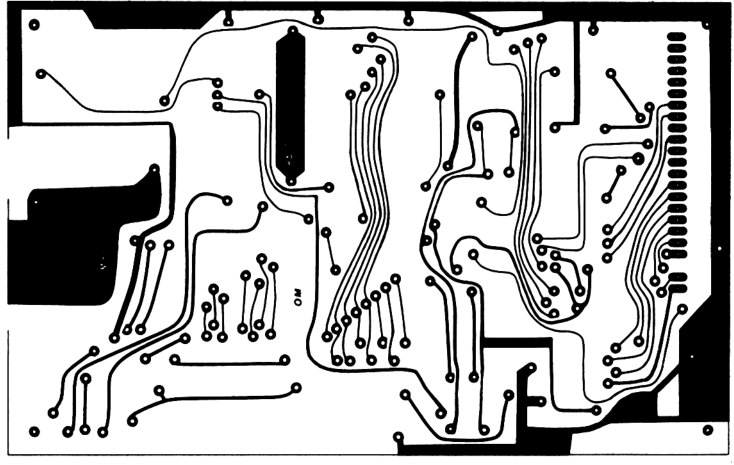
DIL 1		DIL 2	(Ausgabe)
1	Takt (∅)	1	D 0
2	$\overline{\text{RD'}}$	2	D 1
3	$\overline{\text{WR'}}$	3	D 2
4	23 552	4	D 3
5	23 553	5	D 4
6	23 554	6	D 5
7	23 555	7	D 6
8	23 567	8	D 7
9	23 566		
10	23 565	9. . .16 Masse	
11	23 564		
12	23 563		
13	23 562		
14	23 561	DIL 3	(Eingabe)
15	23 560	1	D 0
16	23 559	2	D 1
17	D 0'	3	D 2
18	D 1'	4	D 3
19	D 2'	5	D 4
20	D 3'	6	D 5
21	D 4'	7	D 6
22	D 5'	8	D 7
23	D 6'		
24	D 7'	9. . .16 Masse	

⑤ **Portadressen und Stiftbelegung der DIL-Buchsen:** DIL 1 dient dem Anschluß weiterer Hardware, DIL 2 der Dateneingabe und DIL 3 der Datenausgabe

Eine kleine Zugabe: Software-Schalter

Mit den beiden NAND-Gattern G 2 und G 3 ist ein Set-Reset-Flipflop aufgebaut. Der Schaltzustand des Flipflops ist durch zwei Leuchtdioden an den Ausgängen erkennbar.
Durch einen POKE-Befehl auf Adresse 23 556 wird das Flipflop gesetzt, durch einen gleichartigen Befehl auf Adresse 23 557 rückge-

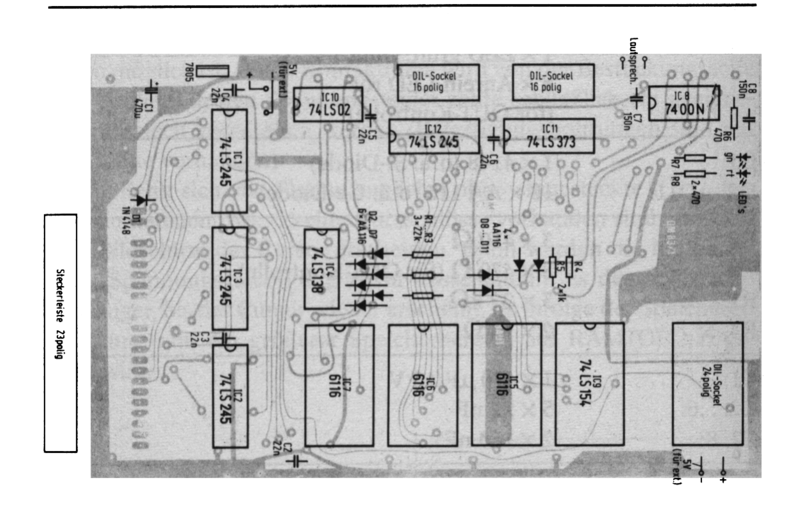




⑥ **Layout der Basisplatine:** Oben die Lötseite und unten die Bestückungsseite der doppelseitig kaschierten Leiterplatte im Europakarten-Format

setzt. Realisiert wird dies, indem die entsprechenden Ausgangsleitungen von IC 9 über die Dioden D 8. . .D 11 mit dem $\overline{\text{WR}}$'-Signal UND verknüpft werden (negative Logik, da L-aktiv). Dieser Schalter kann also, wie auch der Name sagt, von einem Programm aus bedient werden.

Zusätzlich treibt der eine der beiden Schalterausgänge über G 4 einen Lautsprecher. Jedesmal, wenn umgeschaltet wird, ist im Lautsprecher ein Klick-Geräusch zu hören. Erfolgt die Umschaltung schnell genug, entsteht ein Ton. Seine Frequenz ist gleich der Umschaltfrequenz; auf diese Weise ist also eine programmgesteuerte Tonerzeugung möglich. Der Wert von R 6 bestimmt die Lautstärke.
Die verbleibenden 13 decodierten Adreßsignale sind zusammen mit dem Datenbus und dem $\overline{\text{RD}}$’- sowie $\overline{\text{WR}}$’-Signal an die Buchse DIL 1 geführt. Auch der CPU-Takt steht an dieser Buchse zur Verfügung. Hier lassen sich weitere Zusatzschaltungen anschließen, die in späteren Heften der FUNKSCHAU veröffentlicht werden. Portadressen sowie die Stiftbelegung der DIL-Buchsen sind in *Bild 5* zusammengefaßt.



⑦ **Bestückungsplan:** Blick auf Bauteileseite. Die Lötpunkte der hier scheinbar »in der Luft« hängenden Bauelemente befinden sich auf der Lötseite

Aufbautips und Inbetriebnahme

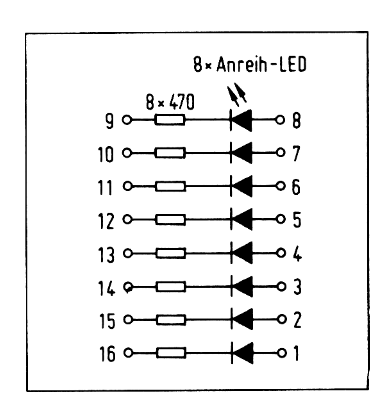
Die Schaltung wird auf einer zweiseitig kupferkaschierten Europakarte aufgebaut *(Bild 6)*. Selbstverständlich ist dabei ein Teilausbau möglich. Durch Weglassen der jeweils nicht benötigten ICs kann die

Platine auf die Speichererweiterung (auch teilweise) oder den I/O-Port bzw. Softwareschalter beschränkt werden. Ein Nachrüsten ist jederzeit möglich.

Stückliste der Gesamtschaltung

IC 1, 2, 3, 12:	4 × 74LS245
IC 5, 6, 7:	3 × HM 6116LP-3
IC 10:	1 × 74LS02
IC 9:	1 × 74LS154
IC 4:	1 × 74LS138
IC 11:	1 × 74LS373
IC 8:	1 × 7400 N
	1 × 7805 (Spannungsregler)
	1 × LED rot 3 mm
	1 × LED grün 3 mm
	8 × Anreih-LED rot
	(for OUT-Kontrolle)
D1:	1 × 4148 (o. ä. Si-Diode)
D2. . .D11:	10 × AA 116 (o. ä. Ge-Diode)
R6. . .9	3 × 470 Ω
	8 × 470 Ω (für OUT-Kontrolle)
R1. . .3:	3 × 22 kΩ
R4, 5:	2 × 1 kΩ
C1:	1 × 470 µF/16 V
C2. . .6:	5 × 22 nF
C7, 8:	2 × 150 nF
Sp:	1 × Lautsprecher 50 Ω
	(geeignet auch Postkapsel 32 Ω)
	1 × Steckerleiste 2 × 23pol., Raster 2,54
	1 × DIL-Schalter 8pol.
	1 × DIL-Stecker 16pol.
	5 × Stecksockel 24pol.
	5 × Stecksockel 20pol.
	3 × Stecksockel 16pol.
	2 × Stecksockel 14 pol.
	6 × Steckstifte

Um bei der Platinenherstellung eine exakte Passung zwischen Vorder- und Rückseite zu erhalten, empfiehlt sich das folgende Verfahren: Die Europakarte erhält vor der Belichtung vier Bohrungen (1 mm Ø), jeweils 5 mm von den Kanten entfernt. Auf einem Holzbrett werden dann vier Stifte in einem Rechteck (9 cm × 15 cm) angeordnet und darauf die Platine und die Vorlage aufgesteckt. Hierzu hat die Platinenvorlage in den Ecken vier Lötaugen, jeweils 5 mm vom Rand entfernt, die durchstochen werden.
Bohrungen werden mit 0,8 mm Durchmesser ausgeführt, mit Ausnahme der Anschlüsse für Steckleiste, Siebkondensator und Spannungsregler (1 mm). Anschließend werden alle von der Bestückungsseite aus sichtbaren Bohrungen durchkontaktiert (Ausnahme: Bohrungen der Steckerleiste). Danach erfolgt das Bestücken (ICs auf Sockel) gemäß *Bild* 7.
Beim Einlöten der Steckerleiste ist genügend Abstand zum Platinenrand einzuhalten, damit das Anstecken an den ZX 81 ohne Schwierigkeiten möglich ist. Die A-Kontaktreihe wird von der Bestückungsseite her verlötet. Für den Spannungsregler sollte schließlich ein geeignetes Kühlblech vorgesehen werden (Befestigungsbohrungen sind auf der Platine vorhanden).
Es empfiehlt sich, die Platine zunächst ohne eingesteckte ICs in Betrieb zu nehmen. Wenn keine Kurzschlüsse vorhanden sind, muß auf dem Bildschirm der Cursor erscheinen. Dann werden die ICs eingesetzt (Spannungsversorgung zuvor ausschalten!). Es dauert jetzt etwas länger, bis der Cursor wieder erscheint, da infolge der Speichererweiterung nun ein größerer Speicherbereich auf RAMTOP hin geprüft wird.



⑧ **DIL-Stecker mit LEDs:** In die Buchse DIL 2 gesteckt, zeigen die LEDs den Binärcode einer mit POKE eingegebenen Zahl

RAMTOP (PRINT PEEK 16388+256*PEEK 16389) liefert bei richtiger Funktion 23552. Falls gewünscht, lassen sich auch nur ein oder zwei Speicher-ICs einsetzen. Dabei ist auf die richtige Reihenfolge zu achten, da der Computer einen durchgehenden Speicherbereich benötigt. RAMTOP ist in diesem Fall 19456 bzw. 21504.
POKE 23556, Zahl schaltet die grüne LED, POKE 23557, Zahl die rote LED ein (Zahl kann zwischen 0 und 255 liegen). Ein an den Lautsprecherausgang angeschlossener Hörer gibt dabei jedesmal ein Klickgeräusch ab.
Um den IN-Port zu testen, wird ein Achtfach-DIL-Schalter in den Sockel DIL 3 gesteckt. PRINT PEEK 23558 liefert den Dezimalwert der eingestellten Binärkombination. Zur Überprüfung des OUT-Ports (DIL 2) sind acht Anreih-LEDs geeignet, die zusammen mit den Vorwiderständen in einen 16poligen DIL-Stecker gelötet werden *(Bild 8)*.
Mit POKE 23558, Zahl leuchten die LEDs gemäß dem Binärwert der eingegebenen Zahl.
Damit ist die Platine einsatzbereit. Anwendungsbeispiele und passende Erweiterungsschaltungen folgen in späteren Heften.

Oskar Merker

Zwischen Low und High

Analogschnittstelle (auch für den VC 20)

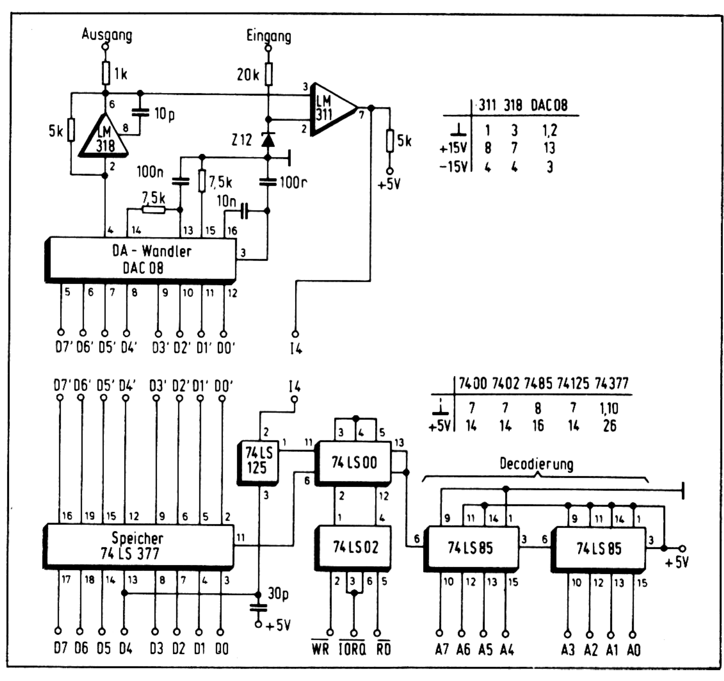
Mit einem A/D- bzw. D/A-Interface wird der ZX 81 zum Generator für beliebige periodische Funktionen oder zum Digitalvoltmeter.

Eine nützliche Erweiterung zum ZX 81 ist ein Analog-Interface, mit dem Spannungen erfaßt und ausgegeben werden können. Für die meisten Anwendungen dürfte eine Auflösung von 8 Bit ausreichen (256 verschiedene Spannungswerte).
Prinzipiell kann man jeden Digital/Analog-(D/A-)Wandler auch als Analog/Digital-(A/D-)Wandler nutzen, indem man die Ausgangsspannung des D/A-Wandlers mit der zu ermittelnden, unbekannten Spannung vergleicht und die Ausgangsspannung solange verändert, bis beide Spannungswerte übereinstimmen. Diese softwaremäßige A/D-Wandlung erfordert nur den geringen zusätzlichen Hardwareaufwand eines Komparator-ICs und einer Eingangsleitung zum Datenbus des Computers.

Der Analogteil liefert +10 V Ausgangsspannung

Die Gesamtschaltung des Interfaces *(Bild 1)* läßt sich in zwei Funktionsgruppen unterteilen: den Analog- und den Digitalteil.
Der obere Teil des Bildes zeigt den Schaltplan des Analogteiles. Als D/A-Wandler findet der preisgünstige 8-Bit-Typ DAC 08 Verwendung. Er wird im »Strom-Ausgangsbetrieb« benutzt, um kürzere Anstiegszeiten zu erzielen. Der schnelle Operationsverstärker LM 318 setzt dann den Ausgangsstrom in die Ausgangsspannung um. Mit dem 5-kΩ-Rückkopplungswiderstand wird am Ausgang ein Spannungsbereich von 0 bis 10 V eingestellt (D0'. . .D7' = »0«: U = 0 V; D0'. . .D7' = »1«: U = 10 V). Durch Ändern dieses Widerstandswertes läßt sich der Ausgangsspannungsbereich verkleinern oder vergrößern ($U = R \cdot 2$ V, R in kΩ). Der 10-pF-Kondensator dient zur Frequenzkompensation und begrenzt die Anstiegsgeschwindigkeit auf 5 V/µs. Durch den 1-kΩ-Ausgangswiderstand ist die Schaltung dauerkurzschlußfest.

Den Vergleich der Eingangsspannung mit der Ausgangsspannung des D/A-Wandlers übernimmt der Komparator LM 311. Ein 20-kΩ-Vorwiderstand und die 12-V-Zenerdiode schützen seinen Eingang vor positiven und negativen Überspannungen. Der zulässige Eingangsspannungsbereich ist gleich dem Ausgangsspannungsbereich, da beide Spannungen unmittelbar miteinander verglichen werden. Am Ausgang I4 des Komparators liegt 0 V (logisch 0), wenn die Ausgangsspannung U des D/A-Wandlers größer als die Eingangsspannung U_x ist; I4 führt + 5 V (logisch 1), wenn U kleiner oder gleich U_x ist.
Der Digitalteil sorgt dafür, daß nur die für den D/A-Wandler bestimmten Daten auch an diesen weitergeleitet werden, und daß beim Abfragen des Komparators dessen logischer Pegel auf eine der acht Datenbusleitungen gelegt wird.



① **Analog-Interface für ZX 81:** Oben: D/A-Wandler mit Komparator zur softwaremäßigen A/D-Wandlung. Unten: Digitalteil zur Port-Adressierung. Damit der D/A-Wandler nicht durch undefinierte Eingangspegel beschädigt wird, sollte der ZX 81 immer eingeschaltet sein, wenn die Spannungsversorgung des Analogteils ein- oder ausgeschaltet wird

Der Digitalteil decodiert die Port-Adresse

Zur Lösung dieser Aufgabe gibt es zwei Möglichkeiten: Erstens könnte man den Eingang des D/A-Wandlers als bestimmten, fest adressierten 8-Bit-Speicherplatz ansehen, in den das Datenbyte zur Ausgabe eingeschrieben werden müßte (POKE) und aus dem die Komparatorinformation auszulesen wäre (PEEK). In diesem Fall sollten alle sechzehn Adreßbits decodiert werden, und es sind, je nach Lage des ausgewählten Speicherplatzes im Adreßbereich, Konflikte mit tatsächlich vorhandenen Speicherzellen zu beachten (Doppelzugriff).
Die elegantere Lösung, die auch durch den Z-80-Befehlssatz unterstützt wird, ordnet dem D/A-Wandler eine bestimmte Portadresse zu ($\overline{\mathrm{IORQ}} = 0$). In diesem Fall sind nur die acht niederwertigen Adreßbits zu decodieren und Speicherplatzkonflikte sind ausgeschlossen. Nachteilig ist hierbei, daß nicht alle Portadressen zugelassen sind, weil das ZX-81-Betriebssystem einige davon belegt, und daß die Bildschirmausgabe während des Zugriffs auf den D/A-Wandler unterbrochen ist.
Die Decodierung der fest eingestellten Portadresse (hier $111 = 2^6 + 2^5 + 2^3 + 2^2 + 2^1 + 2^0$) erfolgt mit zwei 4-Bit-Komparatoren 74LS85: Am Ausgang 6 des zweiten ICs erscheint genau dann eine logische 1, wenn alle acht Adreßbits (A0. . .A7) mit den eingestellten Eingangsbits übereinstimmen, wenn also die gewünschte Adresse am Adreßbus anliegt.
Weiterhin muß die Schaltung erkennen können, ob ein I/O-Port angesprochen wird ($\overline{\mathrm{IORQ}} = 0$) und ob Daten vom Datenbus gesendet ($\overline{\mathrm{WR}} = 0$) oder empfangen ($\overline{\mathrm{RD}} = 0$) werden. Dies wird durch die Verknüpfung der drei Steuerbus-Signale mit dem Ausgangssignal des Adreßbusdecodierers erreicht: Am Ausgang 6 des 74LS00 liegt nur dann eine logische 1, wenn die eingestellte Port-Adresse auf dem Adreßbus liegt und $\overline{\mathrm{IORQ}}$ und $\overline{\mathrm{WR}}$ aktiv (logisch 0) sind. In diesem Fall übernimmt das Speicher-IC 74LS377 das vom Datenbus gesendete Byte B (B zwischen 0 und 255), das dann bis zur nächsten Datenausgabe gespeichert bleibt und vom D/A-Wandler in die entsprechende Ausgangsspannung U umgesetzt wird (U = B/25,5 V).
Bei der Komparatorabfrage muß der Ausgang des LM 311 auf eine der Datenleitungen geschaltet werden. Das Tri-State-Gatter 74LS125 leitet deshalb das Signal I4 an seinem Eingang (2) nur dann an den Ausgang (3) weiter, wenn am Steuereingang (1) logisch 0 liegt. Andernfalls ist der Ausgang hochohmig und vom Eingang entkoppelt. Am Ausgang 11 des 74LS00 liegt das gewünschte Steuersignal an, wenn

die Port-Adresse stimmt und $\overline{IORQ}$ und $\overline{RD}$ aktiv sind; das Komparatorsignal wird dann zur Datenleitung D4 weitergeleitet. Der 30-pF-Kondensator dient zum Abblocken hochfrequenter Störsignale.
Der Digitalteil ist also ein 8-Bit-Parallelausgabeport mit Datenspeicher, der auch alleine als solcher genutzt werden kann. Weiterhin besteht die Möglichkeit, neben der einen für die A/D-Umsetzung benötigten Eingabeleitung die drei übrigen Tri-State-Ausgänge des 74LS125 an den Datenbus anzuschließen und damit 4-Bit-Daten parallel einzugeben.
Drei kleine Demonstrationsprogramme sollen jetzt die Funktionsweise des Interfaces verdeutlichen. Alle Programme laufen auf der Grundversion des ZX 81. Die benötigten Maschinenprogramme werden am Anfang des Programmspeicherbereichs in einem ausreichend langen REM-Statement untergebracht (Zeile 1). Sie sind so vor dem Verschieben im Speicher geschützt und können wie Basic-Programme auf Kassette aufgezeichnet werden.

Ausgabe von Gleichspannung und periodischen Funktionen

Zur Ausgabe eines festen Spannungswertes U muß nur das dem Spannungswert entsprechende Byte B an die Portadresse 111 geschickt werden. Dazu dient folgendes Programm:

```
1 REM ... (hier 9 beliebige Zeichen eintippen)
10 PRINT "AUSGANGSSPANNUNG(V) = ?";
20 INPUT U
30 PRINT U
40 LET B = INT(ABS(U)*25.5 + .5)
50 POKE 16514,B
60 LET A = USR 16515
70 GOTO 10
```

Vor dem Programmstart muß noch das zugehörige Maschinenprogramm aus *Bild 2* (mit POKE Adresse, Code) eingegeben werden. Der auszugebende Bytewert wird zunächst vom Basic-Programm berechnet, in Adresse 16514 abgespeichert (Zeilen 40 und 50) und von dort durch das Maschinenprogramm an die Port-Adresse 111 geschickt (Adressen 16519 und 16520).
Ein zweites Programmbeispiel erlaubt die schnelle Ausgabe von 256 zuvor berechneten Spannungswerten, die alle in den Speicherplätzen

Adresse	Code	Mnemonik
16515	245	PUSH AF
16516	58	LD A,(16514)
16517	130	
16518	64	
16519	211	OUT(111),A
16520	111	
16521	241	POP AF
16522	201	RET

② **Maschinenprogramm zur Gleichspannungsausgabe:** Mit einer Schrittweite von rd. 40 mV lassen sich Spannungswerte zwischen 0 und 10 V einstellen

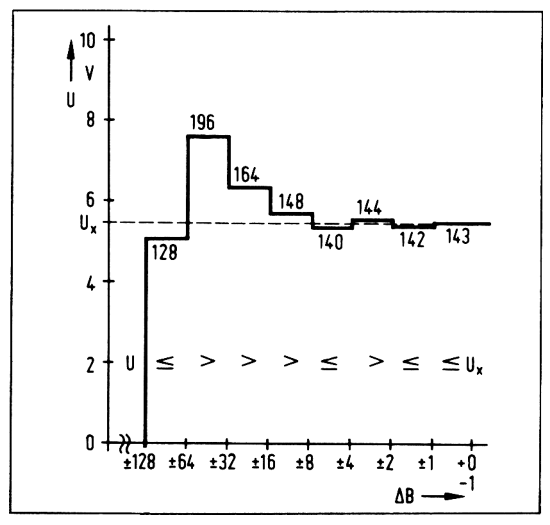
Adresse	Code	Mnemonik
16514	229	PUS HL
16515	245	PUSH AF
16516	197	PUSH BC
16517	14	LD C,111
16518	111	
16519	6	LD B,0
16520	0	
16521	33	LD HL,16896
16522	0	
16523	66	
16524	62	LD A,255
16525	255	
16526	237	OUT(C),A
16527	121	
16528	237	OTIR
16529	179	
16530	62	LD A,0
16531	0	
16532	237	OUT(C),A
16533	121	
16534	193	POP BC
16535	241	POP AF
16536	225	POP HL
16537	201	RET

③ **Funktionswertangabe:** 256 berechnete Spannungswerte (gespeichert ab Adresse 16896) werden periodisch ausgegeben

16896. . .17151 abgelegt worden sind. Durch wiederholten Programmaufruf können so beliebige periodische Funktionen ausgegeben werden, deren Periodendauer nur durch den zur Verfügung stehenden Speicherplatz begrenzt ist. Als Beispiel wird hier eine exponentiell gedämpfte Sinus-Schwingung (Zeile 20) wiederholt ausgegeben (zugehöriges Maschinenprogramm in *Bild 3):*

```
 1 REM . . .24 Zeichen. . .
10 FOR I=0 TO 255
20 POKE 16896+I,(INT(127*EXP(-I/100)*SIN(I/5)+.5)+127)
30 NEXT I
40 LET A=USR 16514
50 GOTO 40
```

Das Programm muß im FAST-Modus gestartet werden, um Unterbrechungen durch die Bildausgabe zu vermeiden. Die Ausgabezeit für die 256 Werte beträgt ca. 1,7 ms (etwa 6,5 µs/Wert).



④ **Softwaremäßige A/D-Wandlung:** Die Ausgangsspannung U des D/A-Wandlers wird mit ständig sinkender Schrittweite ΔB solange verändert, bis der Wert der Eingangsspannung U_x erreicht ist

Analogwerteingabe: Simulation von Meßgeräten

Die softwaremäßige Analog/Digital-Wandlung erfolgt durch schrittweises Annähern der Ausgangsspannung U des D/A-Wandlers an die zu messende Spannung U_x. *Bild 4* verdeutlicht den Meßablauf an einem Beispiel ($U_x = 5{,}61\,V \triangleq B = 143$):
Das Ausgabebyte B wird vom Startwert $2^7 = 128$ (U = 5 V) an jeweils um den Wert 2^n (n = 6. . .0) vergrößert oder verkleinert, je nachdem, ob der Komparatorausgang logisch 1 ($U \leqq U_x$) oder logisch 0 ($U > U_x$) liefert. Nach der vorletzten Abfrage ($\Delta B = \pm 1$) sind ungerade Meßwertbytes wie im Beispiel bereits erfaßt; die letzte Komparatorabfrage ändert B deshalb nur um 0 ($U \leqq U_x$) oder -1 ($U > U_x$).
Würde die Eingangsspannung einen Wert von $U_x = 5{,}57\,V$ (B = 142) haben, so würde bei sonst gleichem Meßablauf letzterer Fall eintreten und $B = 143 - 1 = 142$ als Meßergebnis geliefert.
Das folgende Programmbeispiel simuliert ein Digitalvoltmeter mit einem Meßbereich von 0. . .10 V bei einer Auflösung von 39 mV:

```
 1 REM . . .36 Zeichen. . .
10 LET I = 16514
20 PRINT "ZX 81 ALS DIGITALVOLTMETER", AT 3,5;"U = "
30 PRINT AT 3,7;.01*INT(USR I/.255 + .5);"V"
40 GOTO 30
```

Das zugehörige Maschinenprogramm *(Bild 5)* nimmt die eigentliche A/D-Umsetzung vor, und das Meßergebnis steht nach dem Rücksprung ins Basic-Programm im Ausgaberegister C. Die Messung nimmt im FAST-Modus ca. 200 µs in Anspruch. Das Programm sollte aber wegen der Bildausgabe im SLOW-Modus gestartet werden. Durch das langsame Basic-Programm beträgt die Meßrate dann nur ca. 2,2 Messungen/s.
Sicherheitshalber sollten keine weiteren Basic-Programme im Programmspeicher stehen, da die Gefahr des Überschreibens durch die Meßdaten besteht. Mit Speichererweiterungen ist das nicht zu befürchten, sofern die Daten in einem geschützten Speicherbereich abgelegt werden.
Zum Redaktionsschluß wurden fertig aufgebaute Interfaces vom Autor zu Preisen zwischen 140 DM und 170 DM angeboten.

Manfred Fuchs

Adresse	Code	Mnemonik
16514	245	PUSH AF
16515	229	PUSH HL
16516	6	LD B,128
16517	128	
16518	120	LD A,B
16519	211	OUT(111),A
16520	111	
16521	79	LD C,A
16522	203	SRL B
16523	56	
16524	40	JR Z,14
16525	14	
16526	219	IN A,(111)
16527	111	
16528	203	BIT 4,A
16529	103	
16530	32	JR NZ,4
16531	4	
16532	121	LD A,C
16533	144	SUB B
16534	24	JR-17
16535	239	
16536	121	LD A,C
16537	128	ADD B
16538	24	JR-21
16539	235	
16540	219	IN A,(111)
16541	111	
16542	203	BIT 4,A
16543	103	
16544	32	JR NZ,1
16545	1	
16546	13	DEC C
16547	225	POP HL
16548	241	POP AF
16549	201	RET

⑤ **AD-Umsetzungsprogramm:** Das Ergebnis (Bytewert B) wird im Ausgaberegister C mit ins Basic-Programm übernommen

Schreck in der Morgenstunde

Dieses Mini-Programm *(Bild)* ist hervorragend dazu geeignet, einem nichtsahnenden ZX-81-Besitzer einen Schreck einzujagen. Wird es nach dem Eintippen mit GOTO 50 auf Kassette gespeichert, so erscheint anschließend bzw. nach dem Wiedereinladen ein – je nach Fernseher verschiedenes – dem Ladebild ähnelndes Balkengewirr.
Will man das Programm jedoch mit der BREAK-Taste stoppen, so erscheint plötzlich ein Schirmbild, das schwer nach »Absturz« aussieht. Ursache hierfür ist, daß der POKE-Befehl in Zeile 30 ein Steuerzeichen (K/L-Modus des Cursors) in den Bildspeicher schreibt, das dort eigentlich nichts zu suchen hat und den Bildspeicher durcheinanderbringt. Drückt man NEWLINE, so erscheint das Listing, und der Spuk ist vorüber.
Damit aus dem scheinbaren Absturz kein echter wird, muß das Programm unbedingt genau wie angegeben eingegeben werden. Bei Modellen mit weniger als 3,5 KByte Speicherplatz ist Zeile 30 wirkungslos und der Schreck nur halb so groß! Wolf-Dieter Roth

```
10 SLOW
20 FAST
30 POKE 17000,120
40 GOTO 1
50 SAVE "HUCH"
60 RUN
```

Huch!: Dieses Programm wird ZX-81-Besitzer erschrecken

Die Folie bekennt Farbe

Bevor man darangeht, dem ZX 81 mit einer mechanischen Tastatur Marke Eigenbau auf die Sprünge zu helfen, muß man sich mit der Folientastatur auseinandersetzen.

Die Tastatur des ZX 81 besteht aus drei übereinandergeklebten Folien. Auf der oberen und der unteren sind auf den einander zugewandten Seiten Leiterbahnen und Kontaktpunkte aus einer sehr dünnen Metallschicht aufgebracht. Die mittlere, etwas dickere Folie dient als Trennschicht und hat an den Kontaktpunkten Löcher.
Die Verbindung von der Tastatur zur Leiterbahnplatte des ZX 81 erfolgt ebenfalls durch zwei Folienstreifen mit fünf bzw. acht Leiterbahnen. Die genaue Kontaktbelegung der Tastatur und deren Anschluß an die Leiterplatte zeigt das *Bild.*

Zeilenleitungen teilen die Tastatur

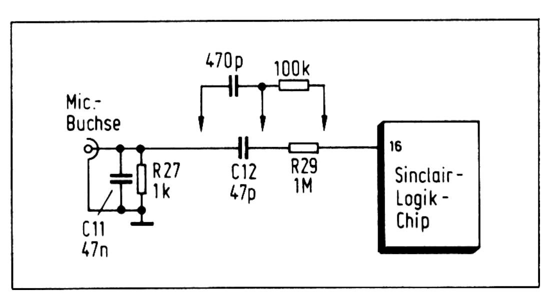
Jede Taste hat die Funktion eines Schließers, wobei stets die blaue und die schwarze Kontaktfläche miteinander verbunden werden. Die fünf Leitungen KBD 0 bis KBD 4 sind dabei die Spaltenleitungen der Tastaturmatrix. Fünf Leitungen genügen, da die Tastatur von den Zeilenleitungen in zwei Hälften geteilt wird (A 11, A 10, A 9, A 8 und A 12, A 13, A 14, A 15). Trotz gleicher Spaltenleitungen für beide Hälften läßt sich so jede gedrückte Taste eindeutig lokalisieren. Die Spaltenleitung KBD 3 z. B. ist für die Tasten 4, R, F, C sowie N, J, U und 7 zuständig.
Wird eine Taste betätigt, so wird die isolierende Luftschicht überwunden, und die Kontaktpunkte berühren sich.

L-Pegel meldet Tastendruck

Ist noch kein Kontakt geschlossen, so liegt an den 5 Anschlußpunkten KBD 0 bis KBD 4 eine Spannung von etwa 5 V an, also der Pegel für logisch H. Dafür sind die fünf Pull-up-Widerstände im Widerstandsnetzwerk RP 3 verantwortlich. An den restlichen Anschlußpunkten A 8 bis A 15 liegt beim Abfragen der Tastatur durch die CPU

	Eingänge					Ausgänge							
Taste	KBD4	KBD3	KBD2	KBD1	KBD0	A11	A10	A12	A9	A13	A8	A14	A15
1	H H	H H	H H	H H	L L	L L	–	–	–	–	L –	–	–
2	H H	H H	H H	L L	L H	L L	–	–	–	–	L –	–	–
3	H H	H H	L L	H H	L H	L L	–	–	–	–	L –	–	–
4	H H	L L	H H	H H	L H	L L	–	–	–	–	L –	–	–
5	L L	H H	H H	H H	L H	L L	–	–	–	–	L –	–	–
6	L L	H H	H H	H H	L H	–	–	L L	–	–	L –	–	–
7	H H	L L	H H	H H	L H	–	–	L L	–	–	L –	–	–
8	H H	H H	L L	H H	L H	–	–	L L	–	–	L –	–	–
9	H H	H H	H H	L L	L H	–	–	L L	–	–	L –	–	–
0	H H	H H	H H	H H	L L	–	–	L L	–	–	L –	–	–
Q	H H	H H	H H	H H	L L	–	L L	–	–	–	L –	–	–
W	H H	H H	H H	L L	L H	–	L L	–	–	–	L –	–	–
E	H H	H H	L L	H H	L H	–	L L	–	–	–	L –	–	–
R	H H	L L	H H	H H	L H	–	L L	–	–	–	L –	–	–
T	L L	H H	H H	H H	L H	–	L L	–	–	–	L –	–	–
Y	L L	H H	H H	H H	L H	–	–	–	–	L L	L –	–	–
U	H H	L L	H H	H H	L H	–	–	–	–	L L	L –	–	–
I	H H	H H	L L	H H	L H	–	–	–	–	L L	L –	–	–
O	H H	H H	H H	L L	L H	–	–	–	–	L L	L –	–	–
P	H H	H H	H H	H H	L L	–	–	–	–	L L	L –	–	–
A	H H	H H	H H	H H	L L	–	–	–	L L	–	L –	–	–
S	H H	H H	H H	L L	L H	–	–	–	L L	–	L –	–	–
D	H H	H H	L L	H H	L H	–	–	–	L L	–	L –	–	–
F	H H	L L	H H	H H	L H	–	–	–	L L	–	L –	–	–
G	L L	H H	H H	H H	L H	–	–	–	L L	–	L –	–	–
H	L L	H H	H H	H H	L H	–	–	–	–	–	L –	L L	–
J	H H	L L	H H	H H	L H	–	–	–	–	–	L –	L L	–
K	H H	H H	L L	H H	L H	–	–	–	–	–	L –	L L	–
L	H H	H H	H H	L L	L H	–	–	–	–	–	L –	L L	–
NL	H H	H H	H H	H H	L L	–	–	–	–	–	L –	L L	–
SH	H H	H H	H H	H H	L L	–	–	–	–	–	L L	–	–
Z	H H	H H	H H	L L	L H	–	–	–	–	–	L L	–	–
X	H H	H H	L L	H H	L H	–	–	–	–	–	L L	–	–
C	H H	L L	H H	H H	L H	–	–	–	–	–	L L	–	–
V	L L	H H	H H	H H	L H	–	–	–	–	–	L L	–	–
B	L L	H H	H H	H H	L H	–	–	–	–	–	L –	–	L L
N	H H	L L	H H	H H	L H	–	–	–	–	–	L –	–	L L
M	H H	H H	L L	H H	L H	–	–	–	–	–	L –	–	L L
.	H H	H H	H H	L L	L H	–	–	–	–	–	L –	–	L L
SP	H H	H H	H H	H H	L L	–	–	–	–	–	L –	–	L L

Tabelle: Den Tasten zugeordnete Pegel auf den 13 Tastaturzuleitungen. Rechte Spalte: ohne gedrückte SHIFT-Taste (auch gültig für F-Modus und G-Modus ohne SHIFT). Linke Spalte: mit gedrückter SHIFT-Taste (auch gültig für G-Modus mit SHIFT)



Höherer Ausgangspegel zum ZX-81: Ein zusätzliches RC-Glied senkt den Wechselstromwiderstand bei gleichbleibender Filtercharakteristik

L-Pegel an. Schließt man nun einen Kontakt, so wird der Pegel an einem der Anschlußpunkte KBD 0 bis KBD 4 auf logisch L heruntergezogen, sobald am entsprechenden Anschluß A 8 bis A 15 ebenfalls logisch L liegt.
Ein Beispiel soll den Signalfluß bei einem Tastendruck verdeutlichen. Gedrückt sei die Taste G: Durch sein Betriebsprogramm wird der ZX 81 gezwungen, etwa alle 400 ms die Tastatur abzufragen, das heißt, er legt kurzzeitig auf eine der Adreßleitungen A 8 bis A 15 L-Pegel (bei gedrückter SHIFT-Taste wird auf zwei Adreßleitungen L-Pegel gelegt). Sobald die Adreßleitung A 9 L-Pegel führt, wird auch die Spaltenleitung KBD 4 auf L-Potential gezogen, da Taste G gedrückt ist. Der Potentialwechsel der Spaltenleitung von H nach L wird vom Sinclair-Logik-Chip (ein von Ferranti im Kundenauftrag gefertigtes Gate-Array) registriert, das nun zusammen mit dem Adreßsignal dafür sorgt, daß im 8-KByte-ROM der Zeichengenerator aktiviert wird. Die dem Zeichen »G« zugeordneten Bitmuster (8 Byte, da ein Zeichen mit einer 8 × 8-Punktmatrix dargestellt wird) gelangen damit auf den Datenbus, und das Zeichen wird am Bildschirm geschrieben.
Eine komplette Zusammenstellung aller möglichen Tastenkombinationen und der daraus resultierenden logischen Pegel an den Anschlußpunkten ist in der *Tabelle* wiedergegeben. Hans-Jürgen Ollech

ZX-81-Hardwaretip:

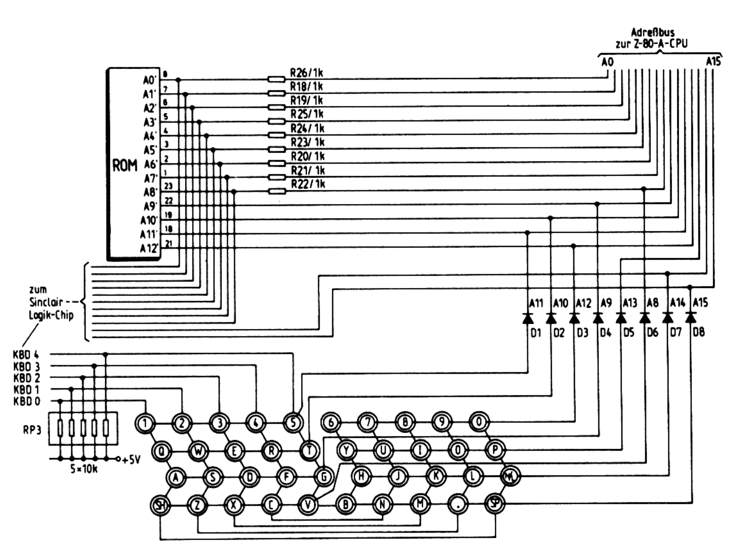
Höherer Pegel bei SAVE

Der beim Abspeichern von Programmen am Mic-Ausgang des ZX-81 auftretende Pegel ist für viele Kassettenrecorder und Tonbandgeräte zu gering, um das Magnetband voll aussteuern zu können. Die Folge sind Aufzeichnungen mit geringer Dynamik und verhältnismäßig hohem Störpegel, die bei einem späteren LOAD Einleseprobleme bereiten.
Abhilfe läßt sich schaffen durch einen kleinen Eingriff in den Computer. Es genügt, parallel zu dem RC-Glied zwischen dem Sinclair-Logik-Chip und der Mic-Buchse (im Schaltplan R 29 bzw. C 12) ein weiteres RC-Glied mit kleinerem Wechselstromwiderstand zu schalten (bewährt hat sich die Kombination von 100 kΩ und 470 pF). Das *Bild* macht dieses deutlich; es müssen lediglich an drei Punkten Lötungen

vorgenommen werden. Auch wenn man keinen Schaltplan des ZX-81 besitzt, lassen sich C 12 (47 pF) und R 29 (1 MΩ) schnell finden, indem man die Leiterbahnen verfolgt, die von der Mic-Buchse ausgehen.

Durch die beschriebene Maßnahme wird die Signalform beim Abspeichern nicht verfälscht, der Pegel jedoch deutlich angehoben, so daß zum Beispiel auch ein Recorder mit DIN- oder Cinch-Eingangsbuchse zur Programmaufzeichnung verwendet werden kann.

Michael Schramm



ZX-81-Tastatur: Ein Tastendruck verbindet die Kontaktflächen. L-Pegel auf einer der Adreßleitungen A 8 bis A 15 ist dann auch für die betroffene Spaltenleitung KBD 0 bis KBD 4 gültig

Kein RAM für alle Fälle

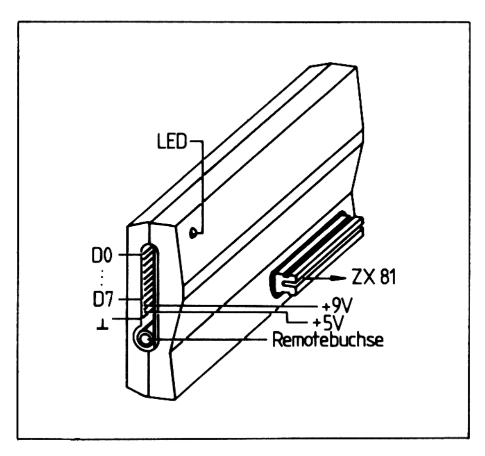
Wer sich für seinen ZX 81 ein 64-KByte-RAM zulegen möchte und glaubt, dann ebensoviel Speicherplatz für Basic-Programme zur Verfügung zu haben – der wird sich wundern.

64-KByte-Speichererweiterungen sind für den ZX 81 in verschiedenen Ausführungen erhältlich. Außergewöhnlich an der hier vorgestellten ist, daß sie, laut Anbieter (Jürgen Schumpich, Ottobrunn), »... volle 64 KByte-Speicher für Basic zur Verfügung stellt...« und »... eine 8-Bit-Parallelschnittstelle für Datenverkehr...« hat. Zusätzlich sollen »... zwei Remote-Buchsen zur Motorsteuerung von Kassettenrecordern...« vorhanden sein.

Für Basic stehen nur 32 KByte bereit

Geliefert wird dieses RAM-Modul *(Bild)* mit einer Bedienungsanleitung, die nicht einmal eine DIN-A4-Seite ausfüllt, und die, zumindest für Einsteiger, an verschiedenen Stellen ergänzungsbedürftig ist.
Die Speichererweiterung verfügt über einen eigenen Spannungsstabilisator, so daß der des ZX 81 nicht stärker belastet wird. Zur Inbetriebnahme wird sie an den ausgeschalteten ZX 81 angeschlossen. Nach dem Einschalten stehen dann für Basic 16 KByte zur Verfügung. Um 32 KByte nutzen zu können, ist es nötig, die Variable RAMTOP auf einen 48 KByte entsprechenden Wert heraufzusetzen (16 KByte gehen durch das ROM und das ROM-Doppel verloren):
POKE 16389,192
Wenn nun noch NEW eingegeben wird, stehen 32 KByte Speicher für Basicprogramme bereit. Der Versuch, höhere Werte einzugeben, scheitert am Betriebssystem, das den Wert von RAMTOP nach Eingabe des Befehls NEW wieder auf den 48 KByte entsprechenden Wert heruntersetzt.
Der verbleibende Bereich von 48 KByte bis 64 KByte ist nur durch PEEK und POKE ansprechbar. Dieser 16 KByte umfassende Adreßbereich läßt sich softwaremäßig umgeschalten, so daß zwei voneinander unabhängige Speicherbereiche mit je 16 KByte bereitstehen. Als Umschalter dient die Speicherzelle 16417, deren Datenbit 0 den einen

oder den anderen Speicherbereich aktiviert. An der Rückseite der Speichererweiterung ist jedoch neben der durchgeschleiften ZX-81-Steckerleiste noch ein Stecker vorhanden: Wird er gezogen, so sind die oberen 4 KByte des Speicherbereichs »1« nicht mehr ansprechbar. Dieser Adreßbereich ist für weitere Zusatzgeräte vorgesehen.
Welcher Speicherbereich oberhalb von 48 KByte gerade in Betrieb ist, wird durch eine Leuchtdiode an der Vorderseite des Moduls angezeigt. Die Zuordnung zwischen Speicherbereich und Einschaltzustand geht freilich aus der Anleitung nicht hervor; beim Mustergerät leuchtete die LED immer dann, wenn Speicherbereich »1« anzusprechen war. 32 KByte dieser Speichererweiterung sind somit für Basic, die restlichen 32 KByte nur für Maschinenprogramme nutzbar.
Das Umschalten mittels der Speicherzelle 16417 bedeutet Einschränkungen, da diese Speicherzelle auch als 8-Bit-Schnittstelle verwendet wird (Bild). Will man volle 8 Bit zur Ausgabe bringen, so muß man nicht nur auf die Speicherbereich-Umschaltung verzichten, sondern auch auf die Remotebuchse (eine, nicht zwei!), da diese durch Bit 1 derselben Speicherzelle geschaltet wird.
Die Schnittstelle ist pro Leitung mit einer TTL-Last belastbar, wobei es jedoch vorkommen kann, daß bei zu vielen unter Belastung gleich-



64-KByte-RAM: Dieses Modell bietet 32 KByte für Basicprogramme, 32 KByte (2 × 16 KByte) für Maschinenprogramme, eine Schnittstelle zur Ausgabe von 8 Bit und eine Remotebuchse, mit der der Motor eines Kassettenrecorders per Software ein- und auszuschalten ist

zeitig geschalteten Leitungen der Rechner einfach nicht mehr weiterarbeitet. Hier hilft nur noch Ausschalten – die Daten sind dann natürlich verloren. Daher ist es ratsam, nur jeweils eine LS-TTL-Last zu treiben. Am besten verwendet man ein Treiber-IC, z.B. SN74LS244, und man ist dieser Sorge enthoben. Zur Eingabe von Daten ist die Schnittstelle ungeeignet; das Wort »Datenverkehr« trifft also nicht zu.

Die Remotebuchse ist für 3,5-mm-Klinkenstecker ausgelegt. Die Anleitung schweigt sich zwar über die Verkabelung zwischen Kassettenrecorder und RAM-Erweiterung aus, aber wenn man zwei Leitungen zum entsprechenden Eingang des Recorders führt, so funktioniert die Sache praktisch auf Anhieb. Wenn sich der Motor des Recorders nicht schalten läßt, schafft das Vertauschen der Anschlüsse Abhilfe.

Der Motor des Recorders läuft an, sobald Bit 1 der Speicherzelle 16417 den Wert »1« enthält. Das Anhalten freilich ist problematisch; da muß man manchmal von Hand nachhelfen (Stop-Taste). Jedoch liegt das dann nicht am RAM, sondern am Recorder – im Zweifelsfalle hilft da wieder der vorhin erwähnte Treiber, mit dem man ein Reed-Relais schaltet, was seinerseits den Recorder in Betrieb setzt.

Fazit: Eine Speichererweiterung, mit der man trotz kleiner technischer Hindernisse gut arbeiten und viel experimentieren kann; zum Preis von 298 DM wird mehr geboten als nur Speicher.

Hans-Peter Gramatke

ZX-81-Speichererweiterung:

Huckepack-RAM

2 × 16 KByte = (vielleicht) 32 KByte

Zwei 16-KByte-RAMs vom Typ Memopak ergeben zusammengefügt nur dann 32 KByte, wenn die Codier-Schalter der Speicher richtig eingestellt sind.

Gegenüber der Original-Sinclair-RAM-Erweiterung haben die 16-KByte-RAMs von Memotech den Vorteil, daß die Schnittstelle des ZX 81 durchgeschleift wird, sie also trotz aufgestecktem RAM nicht blockiert ist *(Bild 1)*. Damit sollten sich eigentlich zwei 16-KByte-RAMs ohne nennenswerte Probleme zu einem 32-KByte-Huckepack-RAM verbinden lassen. Leider ist dem nicht so, solange die vierfach DIL-Schalter an der Rückseite der RAMs nicht die richtige Stellung haben, und leider wurden RAMs verkauft, bei denen die Einstellanweisung fehlt.

Der K-Cursor ist gesucht

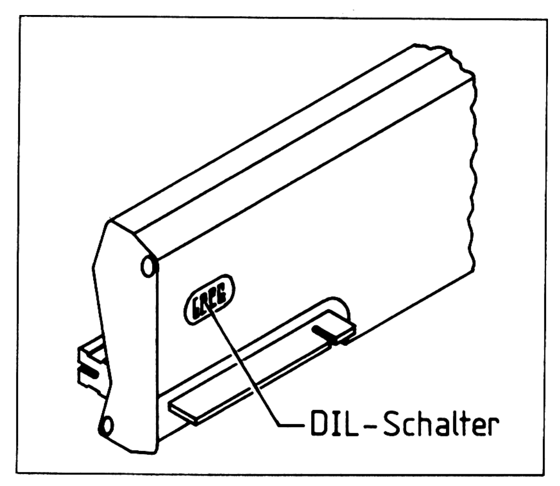
Probeweises Verstellen der Schalter bei eingeschaltetem Gerät führt teils zu einem rhythmisch flimmernden Durcheinander auf dem Bildschirm *(Bild 2 und 3)*, teils erscheint der K-Cursor. In *Tabelle 1* ist gegenübergestellt, welche Wirkungen die 16 möglichen Schalterstellungen bei einem einzelnen Memopak-RAM hervorbringen. Hierbei ist anzumerken, daß Bild 2 nur für einige Sekunden stabil bleibt und anschließend einem weißen Bildschirm Platz macht.
Wenn der K-Cursor erscheint, ist der ZX 81 rechenbereit, und man kann in Erfahrung bringen, ob wirklich 16 KByte Speicherplatz verfügbar sind.

Mit dem Kommando

```
PRINT PEEK 16388 + 256 * PEEK 16389
```

wird Speicherzelle 32768 als Obergrenze (RAMTOP) des RAM-Speichers ermittelt. Laut dem ZX-81-Handbuch liegt die Untergrenze des RAMs auf Adresse 16509. RAMTOP gibt nun nicht die oberste nutz-

bare Adresse an, sondern die erste nicht mehr nutzbare. Daher haben wir 32767 – 16509 = 16258 Byte zur Verfügung – knapp 16 KByte! Hier fallen, ebenso wie in der 1-KByte-Version, die Systemvariablen weg, so daß sich der etwas niedrigere Wert ergibt.



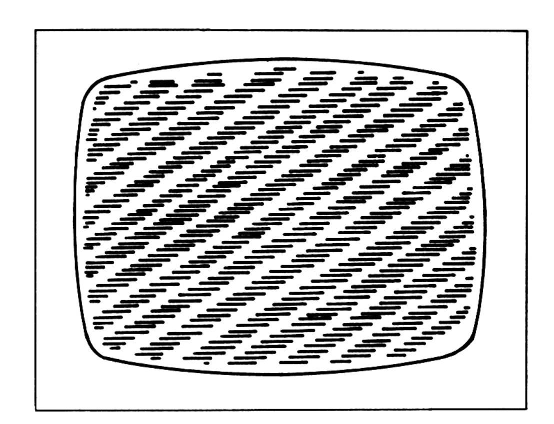
① **16-KByte-RAM von Memotech:** Die Steckerleiste an der Rückseite entspricht der des ZX 81

Zwei Memopaks: Nur 18 Kombinationen führen zum Ziel

Mit zwei 16-KByte-Memopaks gibt es schon 256 Schalterstellungen auszuprobieren. Die Hindernisse sind hier aber noch nicht zu Ende: Denn selbst wenn man Erfolg hat und der K-Cursor erscheint, ist RAMTOP immer 32768.

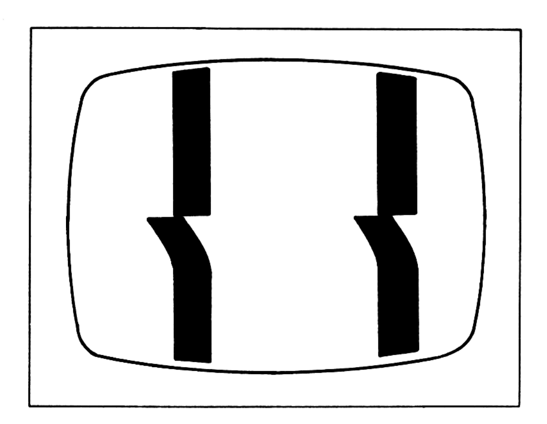
Wie begegnet man nun diesen Widrigkeiten? Zunächst sollte man wissen, daß 151 von den 256 möglichen Schalterkombinationen die verschiedenartigsten Flimmereien auf dem Bildschirm hervorbringen; der ZX 81 ist dabei nicht rechenbereit. Verbleiben also 105 Stellungen, bei denen der K-Cursor erscheint.

Leider aber heißt das noch lange nicht, daß dann auch 32 KByte Arbeitsspeicher verfügbar sind. Denn bei 23 dieser Stellungen hat man trotz zwei Memopaks nur 16 KByte – und mithin nichts gewonnen. In *Tabelle 2* sind diese Kombinationen durch ein »K« gekennzeichnet.



② **Bildsalat:** Zeigt der Bildschirm dieses Muster, ist man auf der falschen Spur

Hierbei gilt, daß unter Memopak 1 die Speichererweiterung zu verstehen ist, die unmittelbar am ZX 81 angeschlossen ist, während Memopak 2 auf Memopak 1 gesteckt wird. Das ist wichtig, weil die Tabelle nicht symmetrisch ist, und damit z. B. die Schalterstellung (6,8) eine andere Wirkung hat als (8,6).
Immerhin gibt es noch andere Varianten: 64 davon bewirken, daß endlich zweimal 16 KByte RAM-Speicher vorhanden sind, jedoch mit einer Lücke von ebenfalls 16 KByte. Die beiden Bereiche liegen zwischen 16 KByte und 32 KByte sowie zwischen 48 KByte und 64 KByte. Zum Rechnen ist freilich wieder nichts, denn das Betriebssystem würde diese Lücke übersehen; dort hineingeschriebene Daten sind selbstverständlich verloren. Tabelle 2 verzeichnet diese Schalterstellungen mit einem »L« (Lücke).
Der nicht gerade große Rest von 18 Schalterkombinationen hat zur Folge, daß tatsächlich 32 KByte Speicher akzeptiert werden – also das gewünschte Ergebnis. In Tabelle 2 ist es mit »S« (Speicher) hervorgehoben.
Der Speicher ist jetzt vorhanden, doch wie nutzt man ihn? RAMTOP nimmt unverdrossen den Wert 32768 an, der aber nur 16 KByte zur Verfügung stellt. Folglich hat der Rechner von den zweiten 16 KByte »keine Ahnung«: und die kann er gar nicht haben!
Beim Einschalten des ZX 81 beginnt nämlich der Z-80-Prozessor, bei Adresse 0 den Programmspeicher zu lesen. Hier ist beim ZX 81 das Betriebssystem (ROM) angeordnet. Dieses veranlaßt den Prozessor, als erstes RAMTOP zu ermitteln. Dazu wird beginnend mit Adresse



③ **ZX-81-Protest:** Diese Darstellung am Bildschirm zeigt's mit Symbolkraft: Der Computer gehorcht nicht mehr

Schalterstellung	Bez.	Wirkung
	0	Bild 2
	1	Bild 3
	2	Bild 3
	3	Bild 3
	4	K-Cursor
	5	K-Cursor
	6	K-Cursor
	7	Bild 3
	8	K-Cursor
	9	K-Cursor
	A	K-Cursor
	B	Bild 3
	C	K-Cursor
	D	K-Cursor
	E	K-Cursor
	F	Bild 3

Tabelle 1: Die 16 möglichen Schalterstellungen (0 bis F) eines Memopak-RAMs und ihre Wirkung. Hier ist der ZX 81 nur dann funktionsbereit, wenn der K-Cursor erscheint

32767 bis hinunter zu Adresse 16384 jede Speicherstelle mit einem Byte geladen. Anschließend werden diese Bytes wieder gelesen und überprüft, ob der zuvor geschriebene Wert vorhanden ist. Die erste Adresse mit abweichendem Inhalt (255 für nicht vorhandene Speicherstelle) wird als RAMTOP interpretiert.
So findet der ZX 81 zwar die oberste Zelle der ersten 16 KByte, aber nicht die Adresse der zweiten. Man muß also RAMTOP selber hochsetzen; der Wert errechnet sich zu $16 \times 1024 \times 3 = 49152$, denn auch hier werden die ersten 16 KByte vom ZX 81 selbst benutzt, so daß nur

Schalterstellung von Memopak 1	Schalterstellung von Memopak 2															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1					K	K	K		L	L	L		L	L	L	
2																
3																
4				K				K				K				K
5	K	K	K		K	K	K		L	L	L		L	L	L	
6	K	K	K	K	K	K	K	K	**S**	**S**	**S**	K	**S**	**S**	**S**	K
7																
8				L				L				L				L
9	L	L	L		L	L	L		L	L	L		L	L	L	
A	L	L	L	L	L	L	L	L	**S**	**S**	**S**	L	**S**	**S**	**S**	L
B																
C				L				L				L				L
D	L	L	L		L	L	L		L	L	L		L	L	L	
E	L	L	L	L	L	L	L	L	**S**	**S**	**S**	L	**S**	**S**	**S**	L
F																

Tabelle 2: Schalterkombinationen für zwei Memopaks. Nur bei den 18 mit »S« hervorgehobenen Kombinationen stehen lückenlos 32 KByte RAM-Speicher bereit

32 KByte übrigbleiben. Folglich muß in Speicherzelle 16388 der Wert »0« stehen (das ist der Fall, wenn man zwei Memopaks angeschlossen hat und den ZX 81 einschaltet). Speicherzelle 16389 verändert man wie folgt:

POKE 16389,192
Zur Kontrolle: $192 \times 256 + 0 = 49152$.

Aber Vorsicht! Allein das Verändern von RAMTOP hat noch nicht die gewünschte Wirkung. Anschließend muß die Anweisung NEW (CLS bleibt wirkungslos!) gegeben werden.
Sollte man nur ein Memopak besitzen, so empfehlen sich die Schalterstellungen A oder E. Ein zweites Memopak kann mit der gleichen Schalterstellung versehen werden, so daß man auf Anhieb 32 KByte Speicher verfügbar hat. Dann braucht man keine Umstellung beim Neukauf des zweiten Memopaks vornehmen und die beiden Speicher dürfen auch vertauscht werden, ohne daß sich etwas am Ergebnis ändert.

Hans-Peter Gramatke

Helfershelfer

Der Handel bietet Programme an, die einem beim Programmieren Hilfestellungen geben. Wir haben zwei dieser Heinzelmännchen für uns werkeln lassen.

Das eine Programm wird unter dem Namen Toolkit (Werkzeug) angeboten. Nach dem Laden, das knapp 2½ min. dauert, versteckt es sich oberhalb von RAMTOP und beansprucht dort rd. 2¼ KByte der unbedingt erforderlichen 16-KByte-Speichererweiterung. Sicher vor dem Programmkiller NEW stehen dann hilfreiche Befehle zur Verfügung.

Toolkit hilft Zeit sparen

Nach dem Laden des Programms meldet sich der ZX 81 mit dem K-Cursor und läßt sich so bedienen, als ob Toolkit nicht vorhanden wäre. Danach darf man ein Basic-Programm eintippen oder eines von Band laden. Außer den üblichen Befehlen kennt der ZX 81 aber nun neun weitere.
RENUM: Wird dieser Befehl erteilt, fragt der ZX 81 danach, welche Nummer die erste Programmzeile bekommen soll, und mit welcher Schrittweite er neu durchnumerieren soll. Das eingegebene Basic-Programm wird dann wunschgemäß umnumeriert. Sprungziele müssen jedoch in Form von Zeilennummern gegeben sein, damit RENUM auch sie aktualisiert (siehe FS 11/83, Seite 77). Dieser Befehl schafft Platz, wenn man Programmzeilen in ein gedrängt numeriertes Programm einschieben will.
DEL: Wie umständlich ist doch das zeilenweise Löschen von Programmteilen beim ZX 81. Mit DEL sind nur die Anfangs- und Endzeilennummer des betroffenen Programmblocks einzugeben – und schon fehlt er beim nächsten Programmlisting.
MEM: Dieser Befehl gibt Auskunft darüber, wieviel Speicherplatz noch frei ist. Damit läßt sich auch die Länge eines eingegebenen Basic-Programms ermitteln.
DUMP: Eine Liste aller zum Zeitpunkt der Befehlsgabe definierten Variablen (außer Feldvariablen und FOR-NEXT-Kontrollvariablen)

wird mitsamt den zugewiesenen Zahlenwerten ausgegeben. DUMP ist z.B. bei der Fehlersuche nützlich.
FIND: Mit FIND läßt sich eine beliebige Zeichenfolge (max. 255 Zeichen) im Basic-Programm aufspüren, egal, ob das z.B. ein Schlüsselwort oder ein Text innerhalb von Anführungszeichen ist. Es erscheint eine Liste aller Programmzeilen, in denen das Suchwort (auch Zahlen sind zulässig) auftaucht. FIND ist bei langen Programmen hilfreich.
REPLACE: Ein z.B. mit FIND gefundener String läßt sich durch REPLACE umtaufen. So ließe sich etwa die Variable EIN überall in einem Programm einfach durch AUS ersetzen, ohne daß dazu das gesamte Programm durchgeackert werden muß. Zahlen können jedoch nicht ersetzt werden, doch darf die Länge des Ersatzstrings von der des ursprünglichen abweichen (statt AUS wäre auch AUSMACHEN zulässig).
REMKILL: Wird der Speicherplatz knapp, werden oft als erstes REM-Zeilen gelöscht, um Platz zu schaffen. REMKILL streicht alle REM-Zeilen auf einmal aus dem Programm.
APPEND: Mit diesem Befehl können zwei verschiedene Basic-Programme im Speicher stehen. Dazu muß das eine Programm mit SAVE (hier Toolkit-Befehl) in einen geschützten Speicherbereich gebracht werden, bevor das zweite Programm von Band zu laden ist. APPEND fügt dann beide zusammen, wobei im Gegensatz zum Programmstapler aus FS 11/83 die Zeilennummern nicht übereinstimmen dürfen (Abhilfe durch RENUM).
Alle Toolkit-Befehle basieren auf Maschinenroutinen und sind daher auch als solche aufzurufen. RAND USR 31850 ruft z.B. die MEM-Routine (mit der Startadresse 31850) auf. Da sich die Adressen schlecht merken lassen, ist alternativ auch der Aufruf RAND USR MEM zulässig. Leider geht der Zusammenhang zwischen Befehlsnamen und Startadressen jedesmal verloren, wenn die Variablen gelöscht werden (z.B. nach RUN). Eine Neudefinition ist zwar möglich, aber der direkte Aufruf mit Startadresse gefiel uns besser – er funktioniert immer. Wie das nachfolgend kurz vorgestellte Programm Screenkit kostete Toolkit zum Redaktionsschluß 28 DM (Profisoft, Osnabrück).

Screenkit wird in Basic-Programme eingebaut

Screenkit kümmert sich hauptsächlich um den Bildschirm, steckt als Maschinenprogramm in REM-Zeilen (nicht sicher vor NEW) und verlangt nach mind 3½ KByte RAM. Es bietet z.B. ein Abräumen des Bildschirms (wie mit einer Schneeschaufel) nach wahlweise einer aller vier Seiten (es lassen sich auch Bildteile wegräumen). Durch Angabe der Eckpunkte lassen sich Rahmen zeichnen, Bildteile löschen und Bildteile invers darstellen.
Ein blinkender Cursor kann an beliebiger Stelle des Bildschirms positioniert werden (warten auf eine Eingabe). Wird dann eine Taste gedrückt, erfolgt der Rücksprung ins Basic, wobei der Code der gedrückten Taste der Variablen X zugewiesen ist. Außerdem lassen sich ausschließlich Variablen auf Band speichern und wieder laden (mit doppelter Geschwindigkeit). Datensätze (z.B. Adreßkartei) lassen sich damit sehr schnell austauschen.
Screenkit erschien uns nicht so elementar nützlich wie Toolkit, zumal ein grundsätzlicher Unterschied zu diesem besteht: Toolkit wirkt so, als ob der ZX 81 dadurch zusätzliche Befehlstasten hätte. Screenkit-Befehle sind dagegen hauptsächlich dann nützlich, wenn sie in ein (neu zu schreibendes!) Basic-Programm eingebaut werden und dann von dort aus die jeweilige Maschinenroutine aufrufen. Screenkit wird mit dem Basic-Programm abgespeichert. Zwar gibt die Bedienungsanleitung ein Verfahren an, wie Screenkit in bereits vorhandene Programme integriert wird, doch das ist eine langwierige Prozedur. -ll

Sachbuch



Taschenbücher



Band 3728
120 Seiten
ISBN 3-426-03728-9

Endlich ein Buch über Home-Computer, das in einer »anwenderfreundlichen Sprache« alles Wissenswerte zu den technischen Wunderwerken aus der Welt der Mikroelektronik bietet.
Die Leistung von Computern, die noch vor einem Jahrzehnt ein Wohnzimmer gefüllt hätten, ist heute in Geräte von der Größe einer Reiseschreibmaschine verpackt, ja, oft sind sie sogar noch kleiner. Am erstaunlichsten aber sind die Preise für die neue Lern- und Spieltechnik, die inzwischen in fast allen Kaufhäusern angeboten werden. Besitzer eines Home-Computers kann man für wenige hundert Mark werden.
Doch wie funktionieren diese Geräte eigentlich, und was ist beim Kauf zu beachten? Welche Anwendungsmöglichkeiten bieten sie, und wie werden sie zu leistungsstärkeren Systemen ausgebaut? Was kann man mit und von Computern lernen, und wie macht man sie zu Helfern im privaten und im kommerziellen Bereich? Kurz, wie kann jeder diese neue Technik optimal für sich nutzen? All das sind Themen dieses Ratgebers.



Sachbuch



Büdeler, Werner: Faszinierendes Weltall
Das moderne Weltbild der Astronomie. 272 S. mit 100 Abb. Band 3700

Berlitz, Charles: Das Atlantis-Rätsel
196 S. mit 15 Fotos, 23 Abb. Band 3561

Berlitz, Charles: Weltuntergang 1999
Droht der Menschheit die Apokalypse?
192 S. Band 3703

Berlitz, Charles/ Manson, Valentine J.: Das Bermuda-Dreieck
Augenzeugen von bisher ungeklärten Phänomenen im Bermuda-Dreieck kommen zu Wort.
216 S. mit 53 Abb. Band 3500

Fischer-Fabian, S.: Die deutschen Cäsaren
Triumph und Tragödie der Kaiser des Mittelalters. 320 S. mit 50 Abb. Band 3606

Fischer-Fabian, S.: Die ersten Deutschen
Der Bericht über das rätselhafte Volk der Germanen.
319 S. mit 50 Abb. Band 3529

Fischer-Fabian, S.: Preußens Gloria
Der Aufstieg eines Staates.
352 S. mit 31 Abb. Band 3695

Fischer-Fabian, S.: Preußens Krieg und Frieden
Der Weg ins Deutsche Reich.
320 S. Band 3720

George, Uwe: In den Wüsten dieser Erde
Ein packender Report über die Geheimnisse der Wüste.
432 S. Band 3714

Kersten, Holger: Jesus lebte in Indien
Kersten verfolgte die Spuren Jesu und kommt zu sensationellen Schlüssen.
280 S. Band 3712

Kerremans, Chuck/ Kerremans, Marlies: Wundern inbegriffen
Die Weltwunder unserer Zeit.
223 S. Band 3694

Ogger, Günter: Kauf dir einen Kaiser
Die Geschichte der Fugger.
352 S. Band 3613



Sagan, Carl: Signale der Erde
Unser Planet stellt sich vor.
301 S. mit 320 z.T. farb. Abb. Band 3676

Sagan, Carl: ...und werdet sein wie Götter
Das Wunder der menschlichen Intelligenz.
272 S. mit 81 Abb. Band 3646



Band 3723
240 Seiten
mit zahlreichen
Abbildungen
ISBN 3-426-03723-8

Zwei Dinge sind es, die Südoldenburg, im Norden unseres Landes gelegen, auszeichnen: Nirgendwo in Deutschland sind so viele Mercedes-Benz-Limousinen zugelassen wie hier, was zumindest verrät, daß man sich in diesem Landstrich aufs Geldverdienen versteht, und nirgendwo sonst auf der Welt wird die Tierproduktion so intensiv betrieben wie hier, was verrät, womit das Geld verdient wird. Doch die alte Weisheit, daß Geld nicht stinkt, stimmt in Südoldenburg schon lange nicht mehr, denn hier stinkt es fürchterlich, tagaus, tagein.
Südoldenburg als Zukunftsmodell unserer Landwirtschaft – ein Beispiel, das jeden das Fürchten lehren kann.

Hunderttausende byte-begeisterte Laien haben mit dem ZX 81 ihre ersten Schritte in die faszinierende Welt der Computertechnik gewagt. Doch nur die wenigsten von ihnen wissen, wieviel Geheimnisse wirklich in dieser kleinen Wundermaschine stecken.

Wer dieses Buch durchgelesen hat, wird zwar noch immer keinen Elefanten von der Bühne verschwinden lassen können, doch wird er seinen ZX 81 so beherrschen, daß es Zauberei gleichkommt. Erfahrene Redakteure und Mitarbeiter der bekannten Zeitschrift *Funkschau* haben hier einen kompletten und leichtverständlichen Kurs zum Programmieren in Maschinensprache zusammengestellt, der mit vielen praktischen Beispielen gewürzt ist und das Erlernen der Sprache zum aufregenden Abenteuer macht.

Knaur®

ISB N 3-426-03794-7 DM +008.80

T 3-28-99



880 ZAUBERN MIT DEM ZX 81

Knaur 3794


AMSTRAD
CPC
OCR
MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA
ACME
https://acpc.me/